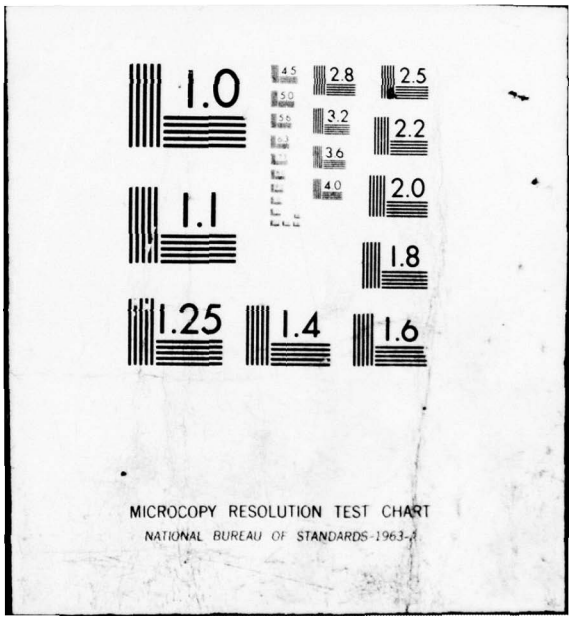
AD-A049 482 ARMY ELECTRONICS COMMAND FORT MONMOUTH N J F/G 9/2
COMPUTER FAMILY ARCHITECTURE SELECTION COMMITTEE FINAL REPORT. --ETC(U)
SEP 77 W E BURR, S H FULLER, P S SHAMAN

UNCLASSIFIED ECOM-4528 NL

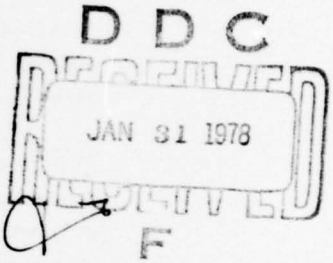
1 OF 2
AD
A049 482


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Research and Development Technical Report

ECOM - 4528

COMPUTER FAMILY ARCHITECTURE SELECTION COMMITTEE - FINAL REPORT, VOLUME III - EVALUATION OF COMPUTER ARCHITECTURE VIA TEST PROGRAMS



William E. Burr
Center for Tactical Computer Sciences
Communications/Automatic Data Processing Laboratory

Samuel H. Fuller
Paul S. Shaman
David A. Lamb
Carnegie-Mellon University

September 1977

ECOM

US ARMY ELECTRONICS COMMAND FORT MONMOUTH, NEW JERSEY 07703

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

The citation of trade names and names of manufacturers in this report is not to be construed as official Government indorsement or approval of commercial products or services referenced herein.

Disposition

Destroy this report when it is no longer needed. Do not return it to the originator.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: (14) ECOM-4528
2. GOVT ACCESSION NO.: (9) Research and development technical rept
3. RECIPIENT'S CATALOG NUMBER:

4. TITLE (and Subtitle): (6) Computer Family Architecture Selection Committee Final Report. Volume III. Evaluation of Computer Architecture via Test Programs.
5. TYPE OF REPORT & PERIOD COVERED:
6. PERFORMING ORG. REPORT NUMBER:

7. AUTHOR(s): (10) William E. Burr, David A. Lamb (CMU)
Samuel H. Fuller (DRE/CMU)
Paul S. Shaman (CMU)
8. CONTRACT OR GRANT NUMBER(s):

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Center For Tactical Computer Sciences (CENTACS)
DRSEL-NL-BC
Fort Monmouth, N.J. 07703
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: (16) 1L7 62701 AH92 00109 (17) P1

11. CONTROLLING OFFICE NAME AND ADDRESS
12. REPORT DATE: (11) September 1977
13. NUMBER OF PAGES: 142 Pages (12) 141 P.

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)
CENTACS
DRSEL-NL-BC
Fort Monmouth, N.J. 07703
15. SECURITY CLASS. (of this report): Unclassified
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES
This report consists of a summary and nine (9) volumes. It is the result of joint Army/Navy work. The complete report may be separately published by the Naval Research Laboratories.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Computer Family Architecture, S, M and R measures of Architecture performance ISP, Instruction Set Processor, Processor/Memory Transfers, Test Program Execution Time.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
This volume presents the evaluation of the Computer Family Architecture (CFA) candidate machines via a set of test programs. The measures used to rank the computer architectures were S, the size of the test program, and M and R, two measures designed to estimate the principle components contributing to the nominal execution speed of the architecture. The S, M, and R measurements are discussed in detail in this report, but the following numbers are the most important, summary results of our evaluation process. These results are the composite performance of the candidate architectures for the set of 12 test programs

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

JOB

037620

20. specified by the CFA committee. Each test program was coded from two to four times on each architecture to minimize the effect of programmer variability. An Analysis of Variance procedure was used to determine the overall relative performance of the three computer architectures. Unity indicates average performance and the lower the score on any of the measures, the better the machine handled the set of test programs.

Architecture	S	M	R
PDP-11	1.00	0.93	0.94
IBM S/370	1.21	1.27	1.29
Interdata 8/32	0.83	0.85	0.83

Table 1

In other words, our test program results indicate that the IBM S/370 needs 46% more memory than the Interdata 8/32 to represent the set of test programs (or 21% more than the average of the three architectures) and the PDP-11 is essentially average in its use of memory. Similarly, the PDP-11's ability to "execute" the test programs ranges between 93% and 94% of the average execution time based on the M and R measure, respectively. Across the test programs used in this study, these results show that for all three measures and Interdata 8/32 is the superior computer architecture, followed by the PDP-11, and the IBM S/370.

VOLUME III
EVALUATION OF COMPUTER ARCHITECTURES
VIA TEST PROGRAMS

1. INTRODUCTION

While there are many useful parameters of a computer architecture that can be determined directly from the principles of operation manual, the only method known to be a realistic, practical test of the quality of a computer architecture is to evaluate its performance against a set of benchmarks, or test programs. In Volume II we presented a set of absolute and quantitative criteria that the CFA committee felt provided some indication of the quality of the candidate computer architectures. It is important to emphasize, however, that throughout the discussion of these criteria it was understood that a benchmarking phase would be needed, and that many of the quantitative criteria were being used to help construct a reasonable "prefilter" that would help to reduce the number of candidate computer architectures from the original nine to three or four. As described in Volume II, this initial screening in fact reduced the set of candidate computer architectures to three: the IBM S/370, the PDP-11, and the Interdata 8/32.

The concept of writing benchmarks, or test programs, is not a new idea in the field of computer performance evaluation. For the purpose of the CFA committee, we define a test program to be a relatively small program (100 to 500 machine instructions) that was selected as representative of a class of programs. The CFA committee's test program evaluation study described here must also address the central problems facing conventional benchmarking studies:

a. How is a representative set of test programs selected?
b. Given limited manpower, how are programmers assigned to writing test programs in order to maximize the information that can be gained?

We face an additional problem here because we are evaluating computer architectures, independent of any of their specific implementation. In other words, when evaluating particular computers, time is the natural measure of how fast a test program can be executed. However, a computer architecture does not specify the execution time of any instructions. Thus an alternative to time must be chosen as a metric of execution speed.

This volume explains how the CFA committee addressed the above questions and presents the results of the test program evaluation of the three candidate architectures. The next section, Section 2, describes how the 12 test programs used in the evaluation process were selected, and Appendix A gives the actual specifications of the test programs used by the programmers to code the programs for the candidate architectures. Section 3 explains the measures of architecture performance that were used in this study. Section 4 explains how 15 programmers were assigned from six to nine programs each, in order to get a set of slightly over 100 test program implementations that were used to compare the relative performance of the candidate architectures. The principle results of the test program evaluation are presented in Section 5, and Appendix E contains the actual S, M, and and R measurements of all of the test programs. The analysis of variance procedure, and related procedures, used to analyze the test program statistics are also reviewed. We conclude this report with a brief summary and discussion of our experiences and problems, that should prove useful for any further work in this area.

2. TEST PROGRAM SPECIFICATION

A Test Program Subcommittee was appointed at the first CFA committee meeting of 1 and 2 October 1975 and was charged with:

(1) determining the general criteria for test programs
(2) developing characteristics to be tested
(3) specifying specific test programs

Members were:

Bill Burr (Chairman)	USA ECOM
LT Belton Allen	NPGS
Mark Stephens	PMTC
Forrest Summer	NTEC
William McCoy	NSWC, Dahlgren

The Test Program Subcommittee met on 20 and 21 October 1975 to formulate recommendations for the December 1975 meeting of the CFA committee.

The ultimate goal of the subcommittee was to design a procedure that would provide data about the relative efficiencies of the candidate computer architectures, and not data about any particular implementation of an architecture. In the conventional benchmark approach, the performance of a particular computer on representative programs (usually coded in a Higher Order Language common to all the computers to be measured) is measured against time. This would not satisfy test program objectives because it measures an actual implementation of an architecture. The subcommittee decided that the measures of architectural merit must be time independent, and would have to be based on the number of bits of memory required to represent a particular specified algorithm and the number of bits which were transferred between the processor and main memory to execute an algorithm. The next section, Section 3, describes in detail the measures of architecture performance used in this study.

a. Alternative Approaches

A number of alternative test program specifications were considered by the Test Program Subcommittee and the entire CFA committee. The major alternatives are discussed here.

(1) Higher Order Language Test Programs

A tempting proposal was to use test programs written in a Higher Order Language (HOL), but to use time independent measures. This had the advantage of allowing a single HOL source program to be used for all the architectures to be tested. This also would have permitted the use of existing benchmark programs, which were coded in FORTRAN, which were available from several sources (FCDSSA, and NADC), and which were extracted from "real" military systems. One disadvantage of this approach was that no one language, even FORTRAN, was available on all the nine initial candidate architectures and those languages developed for use in tactical military applications (e.g., JOVIAL, CMS-2, CS-4, and TACPOL) were each available on only a few of the candidate architectures. There are FORTRAN IV and COBOL compilers available for each of the three final candidate architectures, however neither FORTRAN nor COBOL are widely used in tactical military applications. The major disadvantage, however, was that there was no practical way to separate the effects of compiler quality from the effects of

architectural efficiency, and the object of the experiment was to measure only the architecture. The HOL approach to the Test program effort was ruled out, because the results would necessarily involve a significant undetermined component, which would be due to variations in the efficiency of compilers which are unlikely to be extensively used in tactical military applications. These unmeasurable compiler effects might well mask genuine differences in the intrinsic efficiencies of the architectures.

(2) Synthetic Programs

Another test program approach proposed the use of synthetic programs. Synthetic programs are highly parameterized programs, written in either assembly or higher-level languages, and designed to provide a "representative" computational and I/O load on a computer system. [Buchholz, 1969]. Synthetic programs do not do any useful computation, but rather are structured as a set of loops, controlled by the input parameters of the synthetic program, that compute, initiate I/O activity, or request operating system services. Synthetic loads have been most sucessful in testing computer systems against varying multiprogramming job mixes and for evaluating system configurations. Synthetic programs were not used here because we were interested in how well an architecture's instruction set could represent and execute specified algorithms.

(3) Code Test Programs in Assembly Language

Using standard (Machine-Oriented) assembly language for the test programs was the obvious alternative to the use of Higher Order languages, but it had several obvious disadvantages. First, each program would have to be recoded for each machine, adding to the effort involved. Moreover, this introduced programmer variability into the experiment, and previous studies have shown programmer variability to be large (variations of factors of 4:1 or more are commonly accepted). Finally, it is much more expensive to code in assembly language than in HOL's, and this would limit the size or number of the test programs. Nevertheless, the committee felt that there were ways to limit, separate and measure these programmer effects, while there was no practical way to limit or separate the effects of compiler efficiency. It was therefore decided that the test programs would, of necessity, be coded in assembly language.

b. Guidelines for Test Programs Specification

The Test Program Subcommittee attempted to establish a strategy for defining and coding the test programs that would minimize the variability due to differences in programmer skill. The strategy devised was as follows:

(1) The test programs would be small "kernel" type programs, of not more than 200 machine instructions. (In the end, a few test programs required more than 200 instructions.) It was felt that only small programs could be specified and controlled with sufficient precision to minimize the effects of programmer variability. Moreover, insufficient resources were available to define, code, test, and measure a significant set of larger programs.

(2) Mr. W. L. McCoy proposed that the programs be defined as structural programs, using a PL/1-like Program Definition Language (PDL) and then "hand

translated" into the assembly languages of the respective architectures. This proposal is essentially the IBM Structured Assembly Language methodology, and it was adopted by the subcommittee. It was recognized that PDL definitions would be impractical for some test programs, specifically those intended to measure interrupt handling.

(3) Programmers would not be permitted to make algorithmic improvements or modifications, but rather would be required to translate the PDL descriptions into assembly language. Programmers were free to optimize their test programs to the extent possible with highly optimizing compilers. This "hand translation" procedure of strictly defined algorithms was expected to reduce variations due to programmer skill.

(4) All test programs except the I/O Interrupt Test Program would be coded as reentrant, position-independent (or self relocating) subroutines. This was believed to be consistent with the best contemporary programming practice and provides a good test of an architecture's subroutine and addressing capabilities.

C. Selection of the Twelve Test Programs

The test program subcommittee then specified a set of 15 proposed test program kernels. These kernels were intended to be broadly representative of the basic types of operations performed in military computer systems. These test programs and the proposed methodology were presented to the CFA Committee at its second meeting on 3 and 4 December 1975. Several additional test program kernels were then proposed by Dr. R. Gordon and Mr. W. L. McCoy. An expanded set of 21 proposed test programs was prepared. This expanded set was then presented to the CFA Committee at its third meeting in March 1976.

At that meeting CFA Committee members were asked to rank the test programs in order of their importance. A composite rating of each of the test programs was made, and it was agreed that the top 12 programs would be the basis of the test program experiment. An ad hoc subcommittee was formed to finalize the detailed specification of the Test Programs. This subcommittee met on 12 March 1976 at NRL. Its members were:

S. Fuller (Chairman)	NRL/CMU
W. Burr	USAECOM
L. Haynes	NSWC, White Oak
W. McCoy	NSWC, Dahlgren
L. Denoia	NUSC, New London
D. Parnas	NRL/Darmstadt

This subcommittee reviewed the 12 test programs designated by the CFA Committee and produced a final revised set of Test Program Specifications. The specifications of the 12 test programs are given in Appendix A and they are briefly described below:

(1) I/O kernel, four priority levels, requires the processor to field interrupts from four devices, each of which has its own priority level. While one device is being processed, interrupts from higher priority devices are allowed.

(2) I/O kernel, FIFO processing, also fields interrupts from four devices, but without consideration of priority level. Instead, each interrupt causes a

request for processing to be queued; requests are processed in FIFO order. While a request is being processed, interrupts from other devices are allowed.

(3) I/O device handler, processes application programs' request for I/O block transfers on a typical tape drive, and returns the status of the transfer upon completion.

(4) Large FFT, computes the fast Fourier transform of a large vector of 32-bit floating point complex numbers. This benchmark exercises the machine's floating point instructions, but principally tests its ability to manage a large address space. (Up to one half of a million bytes may be required for the vector.)

(5) Character search, searches a potentially large character string for the first occurrence of a potentially large argument string. It exercises the ability to move through character strings sequentially.

(6) Bit test, set, or reset tests the initial value of a bit within a bit string, then optionally sets or resets the bit. It tests one kind of bit manipulation.

(7) Runge-Kutta integration numerically integrates a simple differential equation using third-order Runge-Kutta integration. It tests floating-point arithmetic.

(8) Linked list insertion inserts a new entry in a doubly-linked list. It tests pointer manipulation.

(9) Quicksort sorts a potentially large vector of fixed-length strings using the Quicksort algorithm. Like FFT, it tests the ability to manipulate a large address space, but it also tests the ability of the machine to support recursive routines.

(10) ASCII to floating point converts an ASCII string to a floating point number. It exercises character-to-numeric conversion.

(11) Boolean matrix transpose transposes a square, tightly-packed bit matrix. It tests the ability to sequence through bit vectors by arbitrary increments.

(12) Virtual memory space exchange changes the virtual memory mapping context of the processor.

The specifications, written in the Program Definition Language, were intended to completely specify the algorithm to be used, but allow a programmer the freedom to implement the details of the program in whatever way best suited the architecture involved. For example, in the ASCII-to-floating-point benchmark, program J, the PDL specification included the statement:

```
NUMBER ← integer equivalent of characters POSITION to J-1 of A1
         where character J of A1 is "."
```

This description instructs the programmer to convert the character substring POSITION, POSITION +1, ..., J-1 to an integer and store the result in the integer NUMBER. It left up to the programmer whether he would sequence through the

string character-by-character, accumulating an integer number until he found a dot, or perhaps (on the S/370) use the Translate-and-Test (TRT) instruction to find the dot, then use PACK and Convert-to-binary (CVB) to do the conversion. It did forbid him to accumulate the result as a floating point number directly, forcing him to first convert to an integer, and then to floating point.

For another example, the Boolean Matrix Transpose specification, program K, included the statement

```
swap B[I,J] with B[J,I]
```

where B is a tightly-packed boolean matrix. An earlier version of the specification said instead

```
TEMP ← B[I,J]
B[I,J] ← B[J,I]
B[J,I] ← TEMP
```

A strict interpretation of this latter example would force the programmer to do explicit bit fetches and stores, as in the Bit Test, Set, or Reset program, F. The later specification allowed more flexibility.

d. PROCEDURES FOR WRITING, DEBUGGING, AND MEASURING THE TEST PROGRAMS

The test programs were written by 15 programmers at various Army and Navy laboratories and at Carnegie-Mellon University. A set of reasonably comprehensive instructions and conventions were needed to insure that the various programmers produced results that could be compared in a meaningful way. Section 4 of this volume discusses the assignments made to the programmers, and shows how these assignments were made to minimize the distortion of the final conclusions due to variations between programmers. In addition, we also agreed that it was not sufficient to just write the test programs in assembly language. We instructed each programmer that all of the test programs that he wrote had to be assembled and run on the appropriate computer.[1] Appendix B is a copy of the test data that we distributed to the programmers. A test program was defined to be debugged for the purposes of the CFA committee's work if it performed correctly on the test data shown in Appendix B. Also note that some of the input parameters are marked to indicate they are the data that was used when the test program's execution performance was measured.

Appendix C describes the details of the subroutine calling conventions assumed for each of the three final candidate architectures. These calling conventions were used by driver programs in the debugging and testing of the test programs. These calling conventions were designed to make as efficient use of each architecture as possible, consistent with the constraint of requiring the test programs to be reentrant, position-independent subroutines.

---

[1] The exceptions were test programs A, B, C, and L since they all require the use of privileged instructions and it was impractical to require programmers to get stand-alone use of all the candidate machines. In these four cases, an "expert" on a test program was designated and he was responsible for reading in detail all implementations of the test program and returning the test programs to the programmer for correction if he detected any errors.

3. S, M AND R: MEASURES OF AN ARCHITECTURE'S PERFORMANCE

Very little has been done in the past to quantify the relative (or absolute) performance of computer architectures, independent of specific implementations. Hence, like it or not, we had little choice but to define measures of architecture performance for ourselves.

Fundamentally, performance of computers is measured in units of space and time. To put it another way, the resources that are used to solve problems on computers have units of space and time (e.g., computer charges commonly include terms of processor seconds used, primary memory space used, and secondary storage space used). The measures that were used by the CFA Committee to measure a computer architecture's performance on the test program were:

Measure of Space

S: Number of bytes used to represent a test program.

Measure of Execution Time:

M: Number of bytes transferred between primary memory and the processor during the execution of the test program.

R: Number of bytes transferred among internal registers of the processor during execution of the test program.

The remainder of this section will develop exact definitions of these three measures. Terms such as internal register and processor will be given very specific definitions and the constraints on the types of byte transfers will be defined.

All of the measures described in this section are measured in units of 8-bit bytes. A more fundamental unit of measure would be bits, but we faced a number of annoying problems with respect to carry propagation and field alignment that make the measurement of S, M, and R in bits unduly complex. Fortunately, all the computer architectures under consideration by the CFA committee were based on 8-bit bytes (rather than 6, 7, or 9-bit bytes) and hence the byte unit of measurement can be conveniently applied to all these machines.

a. TEST PROGRAM SIZE

An important indication of how well an architecture is suited for an application (test program) is the amount of memory needed to represent it. We define S(i,j,k) to be the number of 8-bit bytes of memory used by programmer k to represent test program i in the machine language of architecture j. The S measure includes all instructions, indirect addresses, and temporary work areas required by the program.

The only memory requirement not included in S is the memory needed to hold the actual data structures, or parameters, specified for use by the test programs. For example, in the Fourier transform test program S did not include the space for the actual vector of complex floating point numbers being transformed, but it did include pointers used as indices into the vector, loop counters, booleans required by the program, and save-areas to hold the original contents of registers used in the computation.

If one or more of the final candidate architectures had substantially different data types (as would have happened if the Bo700 had remained a candidate architecture into the test program phase) we would have had to carefully define the

precision required by the input and output data rather than simply defining the actual format of the data. Fortunately, the IBM S/370, PDP-11, and the Interdata 8/32 all have 8-bit characters, 16-bit integers, 32-bit and 64-bit floating point numbers. The fact that some computer architectures support data types not supported by other architectures is not a problem here (e.g., the IBM S/370 supports decimal numbers and 128 bit floating point numbers and the other candidate architectures do not). Test Programs simply specify the format of their input data structures and, if a candidate architecture does not have instructions to directly support a data type, it provides support via emulation in software.

Given the three architectures being compared, the only point at which there is some complication is with the floating point formats. Even though all three architectures support 32 bit and 64 bit floating point numbers, they use different formats and this leads to slightly different amounts of precision. For example, the IBM/370 and Interdata 8/32 use a base of 16 for their exponent while the PDP-11 uses a base of 2 and, in addition, the PDP-11 uses the "hidden bit" technique to pick up a little more precision. These different formats will result in slightly different degrees of precision in the results [Brent,1973]; pathological cases could be constructed where the 32-bit floating point representation in one architecture will enable an algorithm to converge to a reasonable answer while the 32 bit format of one of the other architectures will not converge, and hence the 64 bit format would be required. Clearly we would have been in more serious difficulty if the PDP-10 with its 36-bit format or the UYK-7 with its 48-bit floating point format (16 bit exponent, 32-bit mantissa) had needed to be evaluated.

b. PROCESSOR EXECUTION RATE MEASURES

In selecting among computer architectures as opposed to alternative computer systems, we are face with a fundamental dilemma: one of the most basic measures of a computer is the speed with which it can solve problems, yet a computer architecture is an abstract description of a computer that does not define the time required to perform any operation. (In fact, it is exactly this time-independence that makes the concept of a computer architecture so attractive!) Given this dilemma, one reaction might be to ignore performance when selecting among alternative computer architectures and leave it to the engineers implementing the various physical realizations to worry about execution speed. However, to adopt this attitude would invite disaster. In other words, although we are evaluating architectures, not implementations, it is essential that the architecture selected yield cost/effective implementations, i.e., the architecture must be "implementable."

The M and R measures defined in this section were developed to measure those aspects of the architecture that will affect the performance of implementations of the architecture.

(1) Processor/Memory Transfers

If there is any single, scalar quantity that comes close to measuring the "power" of a computer system, it is the bandwidth between primary memory and the central processor(s). [Bell and Newell, 1971; CR, 1976; Stone, 1975]

This measure is not concerned with the internal workings of either the primary memory or the central processor; it is determined by the width of the bus (w) between primary memory and the processor and the number of transfers per second the bus is capable of sustaining (see Figure 3-1). Since processor/memory bandwidth is a good indicator of a computer's execution speed, an important

measure of an architecture's effect on the execution speed of the computer system is the amount of information it must transfer between primary memory and the processor during the execution of a program. If one architecture must read or write 2x10**6 bytes in primary memory in order to execute a test program and the second architecture must read or write 10**6 bytes in order to execute the same test program, then, given similar implementation constraints, we would expect the second architecture to be substantially faster than the first.

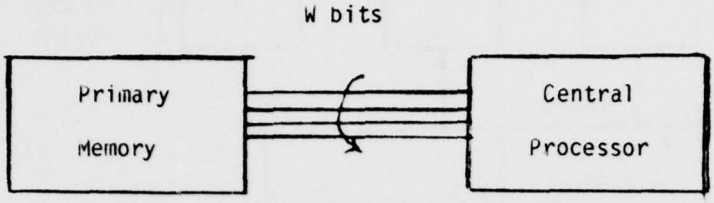


Figure 3-1

The particular measure of primary-memory/central-processor transfers used by the CFA Committee is called the M measure. M(i,j,k) is the number of 8-bit bytes that must be read or written from primary memory by the processor(s) of computer architecture j during the execution of test program i as written by programmer k.

Clearly, there are implementation techniques used in the design of processors and memories to improve performance by attempting to reduce processor/memory traffic, i.e., cache memories, instruction lookahead (or behind) buffers, and other buffering schemes. These are important implementation techniques that are used in high-performance models of any architecture and the structure of the architecture can affect the degree to which these implementation techniques will speed up execution. However, with the intention of keeping our measure of processor/memory traffic as simple, clean, and implementation-independent as possible, none of these buffering techniques will be considered. At the completion of one instruction, and before the initiation of the next instruction, the only information contained in the processor is the contents of the registers in the processor state. (Refer back to Volume II, for the definition of processor state.)

Figure 3-2 may help clarify what I/O traffic is included in the M measure. All the bytes that are transferred to execute a test program across the processor/memory bus and the I/O control bus are included in the M measure.

In many simple computer systems the I/O processors (or I/O channels) of Figure 3-2 are replaced by DMA (direct memory access) controllers or even simpler interfaces under direct (central processor) program control.

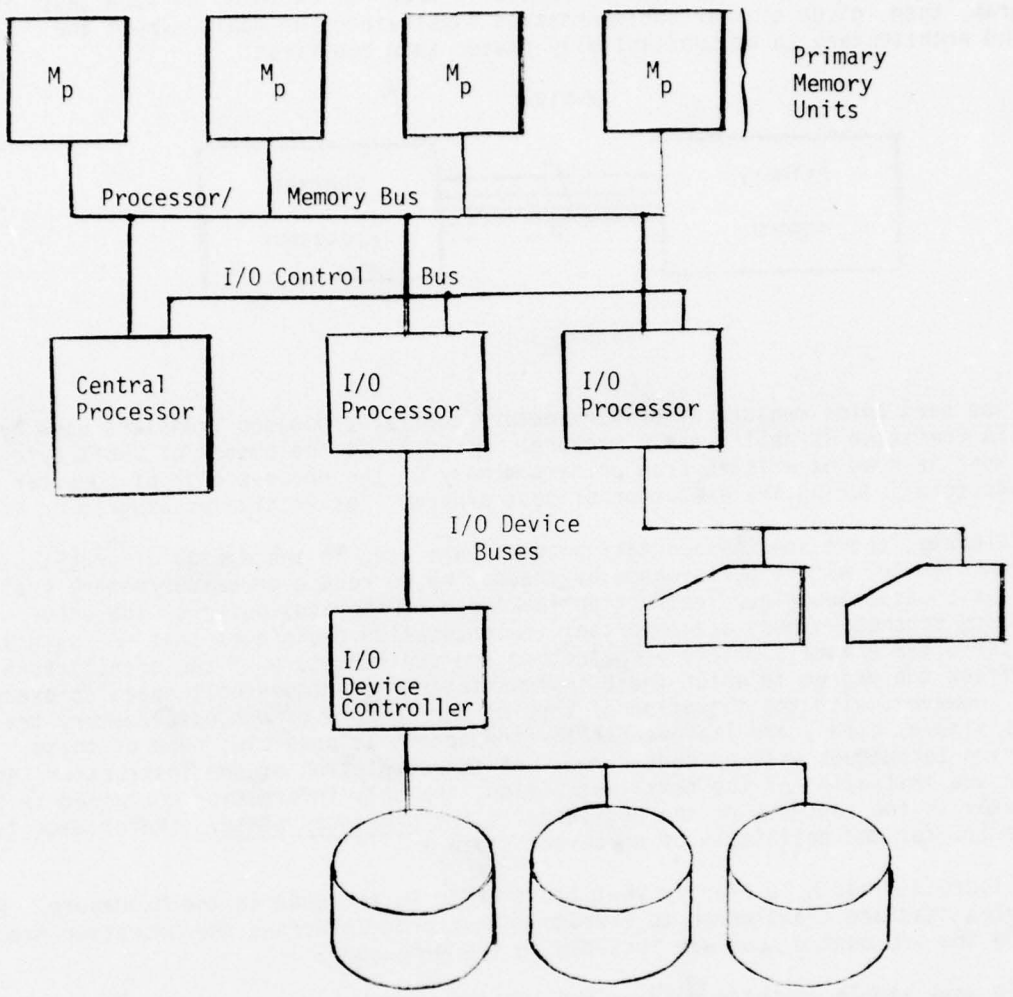


Figure 3-2. Buses of a Computer System

Table 3-1 shows an example of a small IBM S/370 instruction sequence which should help to illustrate the calculation of M. The instructions are the basic loop of a routine for calculating the inner product of two single precision floating point vectors of length 10.

				M
(1)	LA	2,10(0,0)	Set R2 to 16, the length of the vectors	4
(2)	LA	3,XVEC	Load R3 with starting address of X vector.	4
(3)	LA	4,YVEC	Load R2 with starting address of Y vector.	4
(4)	SDR	2,2	Clear floating point reg. 2. Use it to accumulate inner product.	2
(5)	SR	7,7	Clear R7 and use as index into floating point vectors.	2
				1x16= 16
(6) LOOP	LE	4,0(7,3)	Load X(i) into floating point register 4.	8
(7)	ME	4,0(7,4)	Multiply X(i) by Y(i).	8
(8)	ADR	2,4	Sum:=Sum + X(i) * Y(i).	2
(9)	LA	7,4(0,7)	Increment index by 4 bytes.	4
(10)	BCT	2,LOOP	Decrement loop count and branch back if not done.	4
				10x26=260
(11)	STD	2,SUM	Store double precision result in SUM.	12 12
			Total	288 bytes

Table 3-1. M Measure for IBM S/370 Inner Product Example

Instruction (1) is a 32 bit instuction (RX format) and requires no operand fetches from primary memory. Hence the M measure for instruction (1) is 4 bytes. Instructions (2) and (3) are also LA instructions and also add 4 each to the M measure. Instruction (4) is a 16-bit instruction (RR format) and hence M = 2. Note that the fact (4) initiates an 8-byte floating point subtraction does not enter into the calculation of M. (5) is also an RR-format instruction and adds 2 to the M measure. Instruction (6), a load, requires a 4-byte instruction fetch plus a 4-byte data fetch, as does instruction (7). Instruction (8) is a RR instruction which requires only a 2-byte instruction fetch; instruction (9) is another Load Address requiring only a 4-byte instruction fetch, and instruction (10) is a branch requiring a 4-byte instruction fetch. The final instruction, (11), requires a 4-byte instruction fetch and also an 8-byte store to save the double precision floating point sum. Instructions (6) through (10) are each executed 10 times and hence their contribution to M will be the product of the cost of an individual instruction execution and the number of times it is executed. The total M measure of this instruction sequence is 288 bytes.

Although the M measure has been designed to be as simple and clean a measure of processor/memory bandwidth as possible, we discuss below the areas where some clarification is required:

a. Bit and field accessing. Some computer architectures have a set of instructions for manipulating individual bits in memory. In these cases, count in the M measure a one byte read to simply read a bit from memory, and a 1-byte read followed by a 1-byte write to fetch the bit from memory, modify the selected bit and store it back in memory. The following examples, which are described using an ISP-like notation, illustrate this principle:

SBT R1, D(R2)		This is an Interdata 8/32 instruction that sets a bit in memory. D(R2) specifies the beginning of a bit vector and R1 identifies the bit within the vector that is to be set.	
(1)	IR← M[PC:PC+3]	Four bytes of instruction are loaded into the instruction register.	4
(2)	MAR← D+R2<8:31>+ R1<5:28>	Load Memory Address Register with address of byte containing bit to be set.	0
(3)	Rtemp<0:7>← M[MAR] <0:7>	Fetch byte	1
(4)	Rtemp<R1<29:31>>← 1	Set bit	0
(5)	M[MAR]<0:7>← Rtemp <0:7>	Store byte	1
			M= 6 bytes

(Note that several of the steps in the above example do not involve transfers from memory and only transfers within the central processor registers. A measure for processor register transfers will be developed in the next section.)

D. Carry Propagation. Another detail that may cause some ambiguity in the calculation of the M measure is how to handle carry propagation. For example, the PDP-11 INC instruction increments a memory location by 1. How much should be added to the M measure to represent this incrementation? If incrementing on the PDP-11 is counted as a 16-bit fetch followed by a 16-bit store, then this will usually be an overestimate for a PDP-11 implemented with an 8-bit memory bus. With an 8-bit bus, the PDP-11 would fetch 8 bits, increment this value, store the new byte, and only if the carry propagated into the high byte would the second 8 bits need to be modified.

The problem becomes worse when we consider an architecture with 32-bit pointers. If an increment counts as a full word access and then a full word store, we find that incrementing a 32-bit number is much more expensive than a 16-bit number, yet in implementations with memory buses of 16 bits or less, incrementing a 32-bit pointer will almost always take the same time as incrementing a 16-bit pointer.

In order to not overestimate processor/memory traffic with the M measure and to prevent the calculation of the M measure from becoming overly complex, all increments were assumed to require on 8-bit store. Modification of higher order bytes were ignored since they occur infrequently.

(2) Registers Transfers within the Processor

The processor/memory traffic measure just described is our principle measure of a computer architecture's execution rate performance. However, it should not be too surprising that this M measure does not capture all we might want to know about the implementability of an architecture. In this section a second measure of architecture performance is defined: R -- register-to-register traffic within the processor. Whereas the M measure looks at the data traffic between primary memory and the central processor, R is a measure of the data traffic internal to the central

processor. The fundamental goal of the M and R measures was to enable the architecture selection committee to construct a processor execution rate measure from M and R (ultimately an additive measure: aM + bR, where the coefficients a and b can be varied to model projections of relative primary memory and processor speeds). An unfortunate but unavoidable property of the R measure is that it is very sensitive to assumptions about the register and bus structure internal to the processor; in other words, the "implementation" of the processor.

(a) Definition of the R measure

The definition of R is based on the idealized internal structure for a processor shown in Figure 3-3. By using the register structure in Figure 3-3 we do not imply that this is the way processors ought to be built. On the contrary, the structure in Figure 3-3 has a much more regular data path structure than would be practical in contemporary processors. There exist both data paths of marginal utility and non-existent data paths that, if present, could significantly speed up the processor. This structure was selected because the very regular data path, ALU, and register array structure helped simplify our analysis.

R(i,j,k) is defined as the number of 8-bit bytes that are read to and written from the internal processor registers during execution of test program i, as written by programmer k, on architecture j.

The ALU in Figure 3-3 is allowed to perform any common integer, floating point, or decimal arithmetic operation; increment or decrement; and perform arbitrary shift or rotate operations.

(b) Further Detail and Clarification of the R Measure Definition

The definition of the R measure has been the center of considerable discussion with the CFA Committee (see Section 3.4 for chronology of development of S, M, and R). The following discussion and examples are presented here to fully define and clarify the definition of the R measure.

a. Only Data Traffic Measured

All data traffic is measured in R and no control traffic is measured. Figure 3-3 is intended to specify what will be defined to be control traffic and what will be data traffic for the purposes of the R measure. The R measure does not count the following 'control' traffic:

(1) The setting of the condition codes by the ALU (or control unit) and the use of the condition codes by the control unit. The only time that movement of data into or out of the Program Status Word will be counted in the R measure is when a Load PSW instruction is performed or a trap or interrupt sequence moves a new PSW into or out of the PSW register.

(2) Bits transmitted by the control unit to activate or otherwise control the register file, ALU, or memory unit, are not counted in the R measure.

(3) Reading of the Instruction Register by the control unit as it decodes the instruction to determine the instruction execution sequence is not counted in the R measure. In other words, the Instruction Register (with the exception of displacement fields) will be for most practical purposes a write-only register as far as the R measure is concerned.

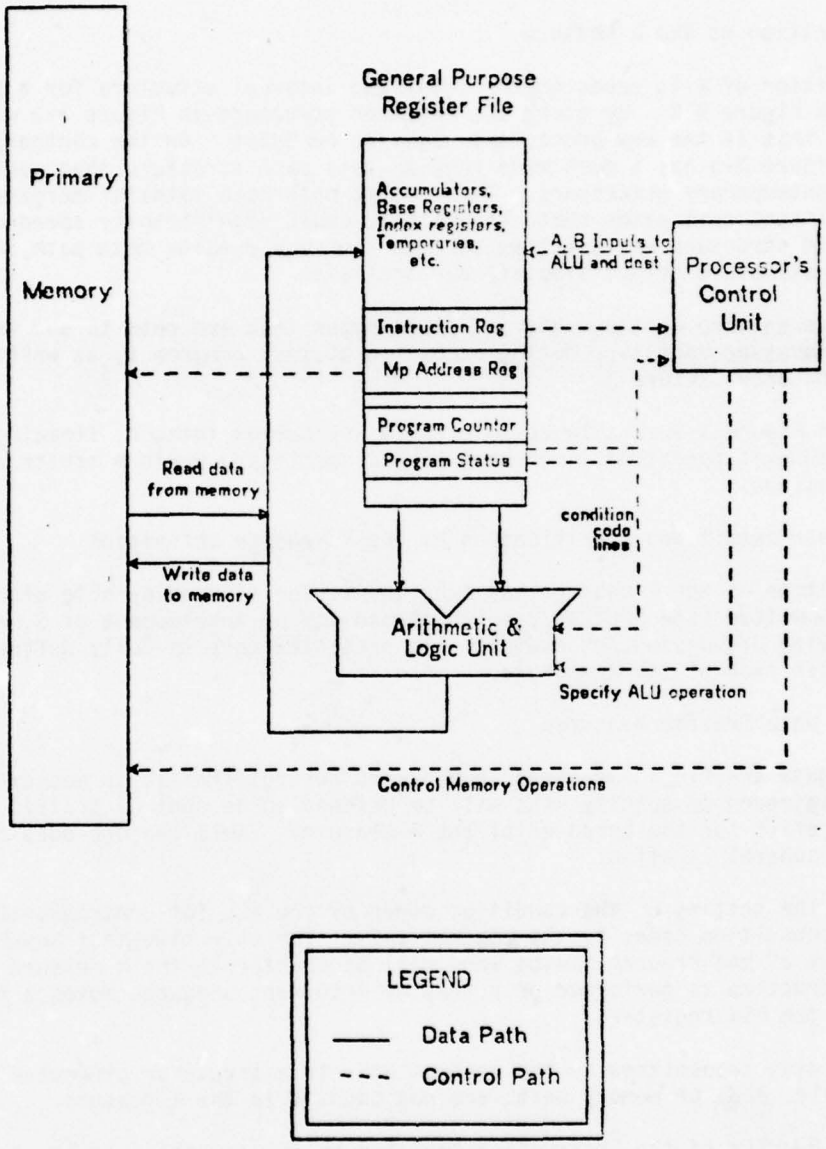


Figure 3-3: Canonical Processor Architecture

(4) Loading the Memory Address Register is counted in the R measure, but use of the contents of the Memory Address Register to specify the address of data to be accessed in primary memory is not counted.

Among data traffic that was a candidate for exclusion from the R measure was the incrementation of the program counter. We decided to leave incrementation on the PC in the R measure, since it is not handled via special circuitry in most minicomputer implementations, and since we have attempted to minimize the number of special "control paths" that we needed to define in our canonical CFA processor structure.

The above four cases of control information excluded from the R measure is meant to be a total enumeration of the control flow in the canonical CFA processor. Because of these control paths, three registers in the general register file (i.e., the IR, MAR, and PS) are specialized in their function. All the rest of the registers in the register file are simply ordinary registers that can present their contents to either the A or B input of the ALU and which can be loaded from the output of the ALU.

b. Virtual Address Translation

The virtual to real address translation process is not counted in the R measure. In other words, the final memory address in the MAR is a virtual address and the work involved in translating this virtual address to a real address is not included in the R measure. There are at least two reasons for this decision.

(1) All practical implementations of processors with a virtual to real translation option do not use the central ALU but in fact use special data paths and adders to allow the memory translation to go on independent of the basic operations in the processor.

(2) The virtual to real translation process will not be used by some members of the CFA.

It should be noted here that the R measure will include the execution of load and store instructions that are used to set up the control registers defining the real to virtual translation process. In other words, the control registers of the memory translation unit are being treated as an "extended PSW" with respect to the R measure.

c. ALU Operations

Assume the ALU in the canonical CFA processor can directly handle all the data types of the architecture: signed integer, logical, floating point, and unsigned integer. The basic reason here is that when performance is of concern the ALU of the processor is capable of handling the appropriate data types. The alternative would be to assume a simple processor (i.e., a two's complement adder) and then emulate the more complex functions such as multiply, floating point operations, etc. The reasons for going for the "full-function" ALU are the following:

(1) When floating point execution time is of concern, processors are used that have floating point ALU's. Floating point operations are emulated on simple two's complement adders when the floating point function is needed but performance of the floating point is not essential.

(2) Calculation of the R measure for floating point operations assuming a two's complement adder would be tedious and given the architectures under consideration there would be little difference in the R measure for the floating point operations of the candidate architectures.

d. Intra-Instruction R Optimization

R measures should be determined for the general case of an instruction (with auxiliary counts to reflect different addressing modes). For example, recognizing that the instruction SUBTRACT R1,R1 is really a CLEAR R1, and then not counting the instruction as a (1) Read R1, (2) Read R1, (3) Write R1, but only (1) Write R1 is not allowed.

Even worse would be to treat as a special case the instruction ADD R1,R2 (where result is stored back in R1) when R2 = 0 and define the R measure to count only the read of R2 to determine it is zero. Any reasonable implementation of an instruction should not try for such optimizations. (Clearly, similar remarks hold for manipulation by zero and one, shifting by zero, etc.)

e. Inter-Instruction R Optimization

The R measure should not be optimized for a test program via inter-instruction optimization. For example, each instruction must fetch its own instruction from primary memory. The idea of assuming an 8-byte or 16-byte IR in the processor and then fetching a new block of instructions only when the current block is exhausted is not allowed.

Similarly, the R measure for an instruction cannot assume a little (or big!) cache in the temporary registers in the register file and only go to primary memory when the quantity is not in the "cache."

f. Consistency between the M and R Measures

There are a number of tradeoffs that were made in the determination of R and M. Many of these tradeoffs were probably acceptable whichever way they were decided, but care had to be taken to be consistent when counting M and R. A good example here was in instruction fetching. The PDP-11, Interdata 8/32 and IBM S/370 all have 2, 4, and 6 byte instructions. If they had been determined by always fetching 6 bytes and throwing away the unused bytes, then it is necessary to always count the instruction fetch as 6 bytes in the M measure. In fact, all three candidate architecture R and M measures were determined by assuming that the machines will only fetch the first two bytes of an instruction and then, based on the operation code, decide if they need to fetch any more bytes from memory. To ensure comparability, one individual, William Burr, computed the M and R measure for each instruction of all three final candidate architectures.

g. Memory Buffer Register

No memory buffer register is specified in Figure 3-3, but any of the general registers in the register file can act as a memory buffer register. (In other words, there is no need for any special control lines to be attached to a MBR.)

h. The Cost of Incrementation

For the purpose of calculating the R measure, incrementation need only involve the low order byte of the register involved. For example, incrementing the

PC by 2 counts as a 1-byte read from the PC and 1-byte write back into the PC. The times when the high order bytes of the PC need to be involved in the incrementation process, because of carry propagation, are sufficiently few that they can be ignored for the purposes of the R measure. The fact that implementation with 16-bit buses will route the low order 16 (or 32) bits of the PC to the ALU for incrementation does not mean the R measure should count a full PC read and write a full PC write. (Our earlier measure of R for 16, 32, 64, etc. bus implementations was designed to model exactly these "inefficiencies" in wide bus implementations.) Another reason for arguing for treating incrementation as less costly than a full word add is that in higher performance machines the PC's will in fact be implemented as a counter, and the lower R count for an increment reflects the simple structure of an incrementer versus a full adder. However, the caution should be repeated here that the R measure bears a stronger relation to simple rather than high performance implementations of the architecture.

i. Constants

Constants of +1 (carry), -1 (borrow), all 0's, or, all 1's are assumed to be "free" (i.e., they don't require access to a register file of constants). All other constants, including 2 and 4, require access to one or more bytes of the constant register file. As a result, incrementing by 1 costs one byte less than incrementing by 2.

j. Shifts

Although Figure 3-3 does not specifically shown them, it is assumed that each architecture has a single register with appropriate lines into the control unit, to control the shift amount. The R measure is not incremented for accessing this register to determine the shift amount; however, it is incremented when that register is loaded.

k. Multiple Assignment Operations not Allowed

In an earlier definition of the R measure, we allowed operations with multiple assignments such as:

```
MAR,PC ← R1.
```

In other words, MAR and PC were simultaneously loaded with the contents of R1. (For a 24-bit MAR and PC this would add up to a total R count of 9 bytes.) While many processor implementations allow such a multiple assignment operation (typically microprogrammed processors termed "horizontal" microprocessors) there are also many implementations that do not allow multiple assignments. After extensive discussions we decided that, since the ISP dialect used by the ISP simulator does not allow multiple assignments, the canonical CFA processor would not allow multiple assignments in order to simplify the collection of R measures.

Hence, the above example must now be split into the following two operations (with a total R count of 12 rather than 9):

```
PC ← R1
MAR ← R1.
```

l. Registers that Must be Present in the Register File of the CFA Processor

There are registers in the canonical CFA processor structure that must be in any implementation of the processor. Clearly all the registers in the processor state of the architecture must be in the register file, since their state must persist across instruction execution (e.g., general purpose register, program status, floating point registers).[2]

m. Registers not in the architecture but in any processor structure

(1) Instruction Register

(2) Memory Address Register: There must be some distinguished register that can present an address to primary memory.

(3) Memory buffer register for read-modify-write operations to primary memory.

(4) Other registers to hold temporary results may be needed by some architectures because of the complex effective address calculation process or because of sequential complexities in the execution of the body of the instruction.

There is no penalty associated with a proliferation of temporary registers in the processor as far as the R measure is concerned. The R measure counts only the number of reads and writes to a register, not the number of registers needed to implement an architecture. Hence, there is a natural tendency in laying out the register file for an architecture to define as many temporaries as there are types of temporary information.

The proposal was made by several CFA members that we limit the R measure to the registers in the architecture of the computer. Specifically, this would mean that the R measure would not count reads and writes to the IR, MAR, memory buffer register, and any other temporary registers in the register file, but not in the architecture. This proposal has the attractive property that it makes the R measure somewhat easier to define (and just as importantly makes the R measure easier to measure automatically on the ISP simulator). The problem with this proposal is that somewhere around 1/3 of the present R traffic is connected to the nonarchitecture registers in the canonical CFA processor. Since the canonical CFA processor is meant to model an implementation of the processor, an R measure restricted to reads and writes of bytes in the registers in the architecture misses about 1/3 of the processor's internal activity.

n. Counting Byte Reads and Writes Rather than Bus Cycles

There was considerable discussion in the CFA committee as to whether the R measure should count the number of (micro)cycles of the canonical CFA processor or the number of bytes read and written from the register file. The decision to count byte reads and writes was made for the following reasons:

(1) The number of bus cycles to execute an instruction is more sensitive to a particular processor structure than the number of bytes that are read and written. Since we would like the R measure to be as "robust" a measure of architecture implementability we have chosen the measure less sensitive to the internal processor bus structure.

---

[2] There exist implementations where the PC state is kept in primary memory not in a register file in the processor, e.g., the TI9900 and the IBM 360/370. In these low-speed implementations much of the R measure must be transferred to the M measure to estimate the true performance.

(2) The practical difference between byte counts and cycle counts is that the processor structure allows two or three byte counts (e.g., two if X ← A and three if X ← A+B) per cycle count. The issue boils down to whether we think it is reasonable to differentiate between these different types of bus cycles. We argue that we should differentiate since a three-byte cycle is more "complex" in either time or to hardware than is a two-byte cycle. Three-byte cycles usually involve an ALU operation, i.e., extra time and three-byte cycles make use of the general structure of the canonical CFA processor -- simple structures such as the PDP-11/20 or PDP-11/40 require two two-byte cycles to get done what the canonical CFA processor gets done in one three-byte cycle.

o. Role of ISP Simulator in M and R Measurement

Both the M and the R measures were measured indirectly by first measuring the number of instructions executed of each instruction type and addressing mode. The number of executions of each instruction type and addressing mode were determined (in most cases automatically via the ARF). The appropriate M and R values for each instruction and addressing mode were then determined from the worksheets given in Appendix D, multiplied by the appropriate factor, and totaled for each program. Although more direct procedures were considered for measuring M and R on the ISP simulator, they were rejected within the context of the time constraints of the CFA committee because:

(1) It was impossible to run some of the test programs on the ARF (e.g., floating point instructions were not implemented in any of the ISP descriptions).

(2) More direct measurements were subject to considerable variation due to the details of the ISP programs which describe the candidate architectures. (See the appendices of Volume IV for the ISP descriptions of the three final architectures.) This variation was substantial because the ISP's were created by different individuals, and because the primary constraint on the description was descriptive clarity, and not efficiency (in the sense of accessing registers the minimum required number of times).

(c) R Measure Calculation.

To simplify the calculation of the R measures, R values for all instructions were considered to have the form:

$$R = R_f + R_a + R_{op} \qquad (3.1)$$

where R_f depends upon the instruction format, R_a is dependent upon the addressing mode of the instruction, and R_{op} is determined by the operation performed.

The calculation of R measures is illustrated in Figure 3-4, which shows how the R count is determined for an IBM S/370 RX format instruction:

A RY,X(R2,R7).

RX instructions are considered to have four distinct addressing modes, depending upon whether the base or index registers specified are zero or nonzero. Using the ARF, and inserting suitable "hooks" in the ISP descriptions, it was easy to count the number of address calculations of each type, the number of executions of each particular instruction format (i.e., RR, RX, SS, etc.), and the number of executions of each operation.

RX, RS, & SI INSTRUCTION INTERPRETATION

	R	COMMENT
	-	-------
IR<0:15> ← Mh[MAR]	2	Get hlfwd in instr reg
MAR ← MAR + 2	3	Inc counts only 1 byte
IR<15:31> ← Mh[MAR]	2	Get rest of instr in IR
PC ← PC + 4	3	Inc Prog Counter
address interpretation	-	
instruction execution	-	
MAR ← PC	6	Set up MAR for next instr.

TOTAL	16	

RX ADDRESS CALCULATION

1. B2 = 0, X2 = 0	R	COMMENT
MAR ← IR<20:31>	5	Just 12 bits from IR
2. B2 = 0, X2 > 0		
MAR ← IR<20:31> +		12 bits plus 24 from reg
R[x2]<8:31>	8	
3.B2 > 0, X2 = 0		
MAR ← IR<20:31> +		
R[B2]<8:31>	8	
4. B2 > 0, X2 > 0		
MAR ← IR<20:31> +		
R[B2]<8:31>	8	
MAR ← R[x2] + MAR	9	12 bit displ plus two regs

TOTAL	17	

EXAMPLE INSTRUCTION

A R4,DISP(R2,R7)	RX ADD INSTR	
		R
RX instruction interpretation		16
address interpretation		17
MBR ← Mw[MAR]		4
R[R1] ← R[R1] + MBR		12
		--
	TOTAL	49

Figure 3-4. IBM S/370 R Measure Example

Figure 3-5 shows a similar instruction for the Interdata 8/32:

```
A   R1,D2(X2)
```

Although the addressing modes for the Interdata 8/32 are somewhat different from the IBM S/370, the example chosen also adds two registers (the PC and X2) plus a displacement to generate the effective address, and the basic structure of the machines are quite similar. Consequently, the corresponding add instruction generates the same R count in both machines.

While calculation of the R measures for the Interdata 8/32 and IBM S/370 is straightforward in most cases, this is not true for the PDP-11. This is because the IBM S/370 and Interdata 8/32 have a rich set of operation codes, which determine the precise instruction format, and which limit the addressing modes to a few similar possibilities. The destination operand of an IBM S/370 add instruction is always a register, and there are only 4 possible (and very similar) source operand addressing modes. Moreover, the source operand is always in memory. The PDP-11 has far fewer operation codes, but a rich set of addressing modes. In general, either the source or destination operand may be a register or a memory location, which is addressed in one of 7 ways. That means that each two operand instruction has a total of 64 combinations of addressing modes. In addition, there are interactions between the operations and addressing modes, which affect the R measure. For example, it makes a difference in the R_{OP} of a MOV instruction if the destination of the MOV is a register or a memory location. It also makes a difference if the destination of an ADD is a register or a memory location. Unfortunately, the effects are different for the ADD and the MOV, because the ADD fetches and stores in the destination, while the MOV only stores in the destination.

The optimum R measure for each of 12x64 combinations of double operand instuction and addressing modes could in principle be computed, as well as the R measure for each of 46x8 combinations of floating point instructions and addressing modes, and roughly 30x8 combinations of single operand instructions and addressing modes. This would have been very tedious, and would have corresponded to an implementation which directly decoded the combination of the OP code, source mode, and destination mode, rather than just the OP code, and then implemented individual microcoded routines for each case.

Of course, such an implementation is possible, as are various compromises which would directly decode special cases of interest, but not every possible case. These cases would not correspond to the kind of simple, regular implementation we have assumed in our development of the R measure. The PDP-11 R measure was calculated assuming that only the operation code was decoded by the control unit. Moreover, the remaining instruction decoding, which is assumed to be performed in microcode, is done in a very regular and straightforward way. Obvious interactions of addressing mode and operation were accounted for within this regular structure, but no attempt was made to optimize every special case, or to recognize and optimize important special cases.

Figure 3-6 illustrates how the PDP-11 R measures were computed. Simply fetching the first word of the instruction and incrementing the PC generates an R measure of 9. A mode 6 address calculation requires an R count of 15. Performing the ADD itself takes an R count of 8, resulting in a total R count of 32. Figure 3-7 illustrates how this same instruction might have been optimized on a case by case basis. The result now is R=28, which is a savings of 12.5% in R.

RX1 AND RX2 INSTRUCTION INTERPRETATION

	R	
IR<0:15> ← Mh[MAR]	2	Get halfwd in instr reg
MAR ← MAR + 2	3	Inc. counts only 1 byte
IR<16:31> ← Mh[MAR]	2	Get rest of instruction
PC ← PC + 4	3	Inc prog counter
address interpretation	-	
instruction execution	-	
MAR ← PC	6	Prepar MAR for next instr
TOTAL	16	

RX2 EFFECTIVE ADDRESS CALCULATION

1. X2 = 0	R	COMMENT
	-	-------
MAR ← IR<18:31> + PC	8	15 bit disp plus prog ctr
2. X2 > 0		
MAR ← IR<17:31> + PC	8	15 bit disp plus prog ctr
MAR ← MAR + R[X2]<8:31>	9	add 3 bytes from reg
TOTAL	17	

EXAMPLE INSTRUCTION

A R1,D2(X2) RX2 ADD INSTRUCTION

	R
RX2 instruction interpretation	16
address interpretation	17
MBR ← Mw[MAR]	4
R[R1] ← R[R1] + MBR	12
TOTAL	49

Figure 3-5. Interdata 8/32 R Measure Example

BINARY WORD OPERATION, S = 0, D = 0

	R	COMMENT
	-	-------
MAR ← PC	4	load MAR for instr fetch
IR ← Mw[MAR]	2	Load instr reg
PC ← PC + 2	3	Inc prog ctr
instruction execution	-	instr type dependent

TOTAL	9	

BINARY OPERATION, S > 0, D = 0

	R	COMMENT
	←	←←←←←←
MAR ← PC	4	Load MAR for instr fetch
IR ← Mw[MAR]	2	Load instr reg
PC ← PC + 2	3	Inc PC
effective addr. calculation	-	Mode dependent, addr in MAR
instruction execution	-	instr type dependent

TOTAL	9	

ADDRESS CALCULATIONS

1. MODE 2

MAR ← R[s/d]	4	Get addr from reg
R[s/d] ← R[s/d] + 2	3	This is 2 for byte operations

TOTAL	7	

2. MODE 6

MAR ← PC	4	Set up MAR to read addr
PC ← PC + 2	3	Inc PC for next instr fetch
MAR ← Mw[MAR]	2	Get addr in MAR, and
MAR ← MAR + R[s/d]	6	add index

TOTAL	15	

EXAMPLE INSTRUCTION

```
------------------
        ADD  X(%2),%1       ;ADD INSTRUCTION, S = 6, D = 0
```

	R
Binary ins interpretation, s > 0, d = 0	9
Source Addr calculation, mode 6	15
MBR ← Mw[MAR]	2
R[s] - R[s] + MBR	6

TOTAL	32

Figure 3-6. PDP-11 R Measure Example

EXAMPLE INSTRUCTION

```
ADD   X(%2),%1                    ;    ADD Instruction, S=6, D=Ø
```

	R	COMMENT
MAR ← PC	4	Load MAR for instr. fetch
IR<0:15> ← Mw[MAR]	2	Load instr. reg.
MAR ← MAR + 2	3	Bump MAR to get
IR<16:31> ← Mw[MAR]	2	rest of instruction
MAR ← IR<16:31> + R[2]	6	Compute source address
MBR ← Mw[MAR]	2	Get source operand
R[1] ← R[1] + MBR	6	Add source and dest
PC ← PC + 4	3	Increment PC
	28	

Figure 3-7. Example of Case by Case R Measure Optimization for the PDP-11

It can certainly be argued that, because the architectures of the IBM S/370 and the Interdata 8/32 fit more naturally into the model of equation 3-1, and because they get to decode an 8 bit operation code which includes some information carried in the mode fields of the PDP-11, that the model chosen is somewhat biased against the PDP-11.

A number of other minor specifications were also made uniformly to the R measure calculations of the 3 architectures. These involved instructions which do different things in different cases. For example, there should properly be a distinction made between conditional branch instructions which do branch, and those which do not. To simplify the collection of data, conditional branches and branch and count type instructions were always assumed to branch.

(d) Range of Applicability of the R Measure

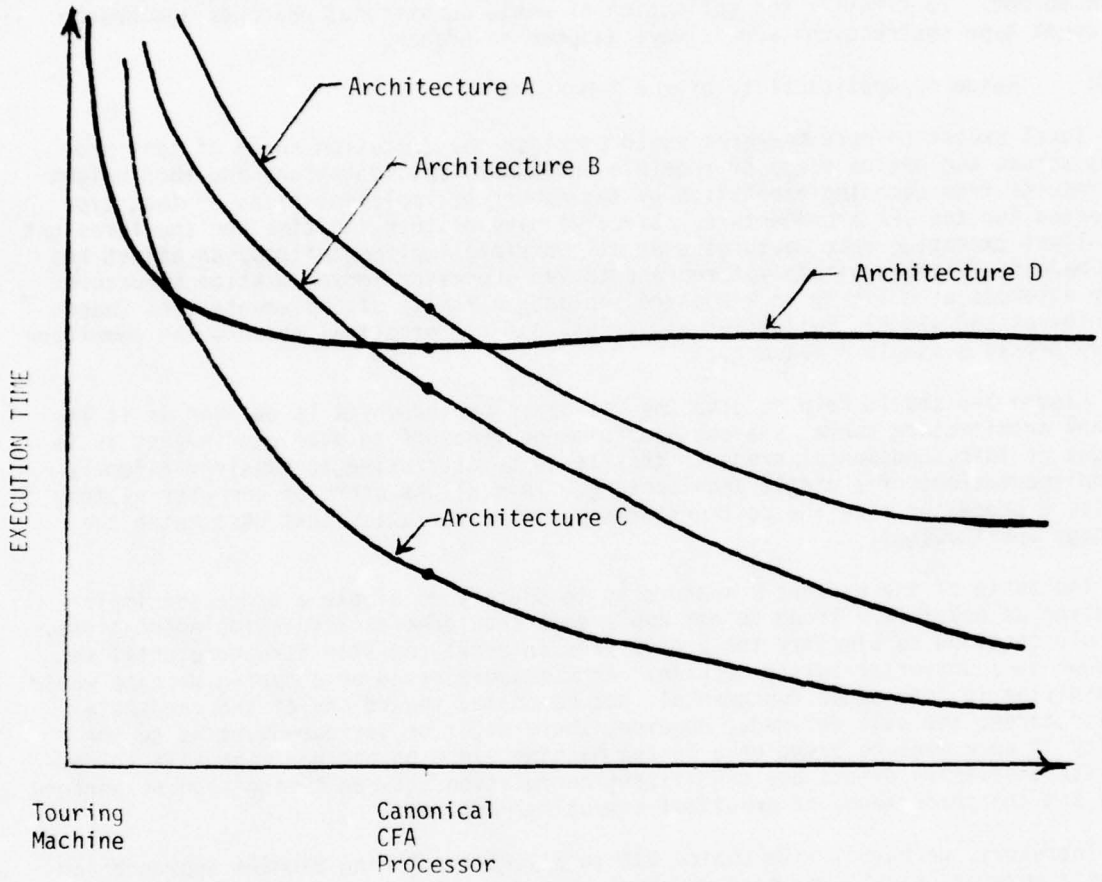
Ideal execution-rate measures would consider the execution speed of test programs across the entire space of feasible processor implementations and then weight the results from each implementation by the number of implementations of that type projected for the CFA architecture. Since we have neither the time nor the foresight to collect execution rate measures over all possible implementations, an effort has been made to define a simple yet representative processor implementation structure. Early attempts at defining an R measure included a family of implementations (based on internal bud width) [Fuller, et al, 1975], but for practical reasons the committee backed off to a single R measure.

Figure 3-8 should help to give the rationale behind why R is defined as it is. For any architecture there is a cost-performance tradeoff to make and in fact it is because of this fundamental tradeoff that it is so attractive to consider a family of implementations of a single architecture. This allows users of computer systems to pick a processor with the cost-performance characteristics that best match the intended applications.

The basis of the present R measure is to specify as simple a processor implementation as possible. If we do not apply some knowledge of actual implementations, we could continue to simplify the processor's internal register structure until we are down to a universal Turing machine. An R measure based on a Turing machine would certainly be in some sense fundamental, not be biased toward any of the candidate architectures, and well defined. However, there might be serious doubt as to the utility of an R measure based on a Turing Machine since no one has been able to demonstrate that there exists any significant correlation between Turing machine performance and the performance of practical computing machines.

Therefore, we have little choice but to reject the Turing machine approach and define a more practical (and more complex) processor structure. This is what we have done with the current R measure. Keep in mind there are undoubtedly many "simple" implementation structures on which it would be appropriate to base the R measure in addition to the specific one selected. However, the one selected is meant to be a moderately simplified model of the internal register structure of contemporary minicomputer processors and microcomputer processors based on bit-slice LSI packages.

A shortcoming of the present R measure (as opposed to the set of R measures originally proposed) is that it is a questionable indicator of high performance implementations. Figure 3-8 attempts to illustrate this. Assuming R is a good indicator of performance for the register structure shown in Figure 3-3, it is probably



* The assumption underlining the use of a scolar R measure is that the candidate architecture will behave similarly to Architectures A, B, and C above, and not like D.

Figure 3-8: The Cost-Performance Space of Processor Implementation

also a good indicator or processor implementations "close" to this canonical structure on the cost-performance curve. However, as we consider high performance implementation, instruction buffers, caches, pipelining, forward cancelling, etc. begin to play an important role in processor performance. The present R measure is of questionable utility in predicting the performance of architectures on these high performance implementations. Given that much of the complexity of high performance implementations is designed to maximize the rate at which the processor can fetch new instructions and data from memory, probably the best practical indicator the committee can use for the relative performance of very high performance processors would be simply the M measure.[4]

c. PROCEDURES FOR COLLECTING DATA AND COMPUTING THE S, M, AND R MEASURES

From the beginning we realized it would not be realistic to give every programmer the definition of S, M, and R and ask him to calculate these measures for his own test programs. An important simplification in the computation of the M and R measures was obtained by tabulating the measures for each instruction (and addressing mode) for the three architectures. The M and R measures for each instruction were then tabulated in the form of a "worksheet." Copies of these worksheets are given in Appendix D.

Given the worksheets, the computation of M and R reduces to counting the number of times each instruction and addressing mode is executed. Two alternative procedures existed for this phase of the computation. An ISP simulator was available (described in Volume IV) that took an ISP description of the computer architecture, the machine language representation of the test program to be executed, and simulated the execution of the test program on the machine described in ISP. The ISP simulator has been instrumented to accumulate the number of times each instruction and addressing mode is executed and hence provides exactly the information needed for computing the M and R measures. The majority of test programs were measured using the ISP simulator. Unfortunately, there were a few test programs that were not easily measured using the ISP simulator and these had to be done by hand. The reasons for measuring some of the test programs by hand were: (1) within the time available we could not get a working version of the test program in a machine-readable format that we could input to the ISP simulator on the CMU PDP-10, or (2) the test program involved too much I/O activity and the ISP simulator was not constructed to handle I/O operations automatically. (However, in the final analysis it would probably have been faster and less error-prone to single-step the ISP simulator through I/O operations interactively and then get instruction counts from the ISP simulator than to do it manually.)

The final step in the S, M, and R computations was handled by another program that performed the additions and multiplications indicated on the architecture worksheets. This program could either read instruction count information directly from the output files generated by the ISP simulator or request the manual entry of instruction counts for those test programs measured by hand. In addition to eliminating many tedious computations, this program eliminated many errors that would have crept into the measures if this final tallying process had been done by hand.

---

[4] For the specific set of 105 test programs written and measured in the CFA project, there is a very strong positive correlation between all the M and R measures. No decision made by the CFA committee would have changed if just the M measure, rather than a weighted sum of the M and R measures, had been used.

Another feature of automating the final step of computing the S, M, and R measures is that if modifications to the S, M, and R measures are required, it is now possible to change just the measures in the program and then rerun the S, M, and R computations for all the test programs. Investigating the effect on the R measure of "microcoded" floating point operations could also be pursued this way.

d. CHRONOLOGY OF DEVELOPMENT OF S, M, AND R MEASURES

Some of the reasons for the particular form the S, M, and R measures have taken are due to the time constraints and the sequence of development of these measures over a period of time ranging from early October, 1975 through July, 1976. The following account traces the development of these measures and attempts to indicate where important decisions and changes were made in the definition of these architecture measures.

An initial memo [Fuller and Smith, 1975] discussing selection criteria was distributed at the initial CFA meeting on 1 and 2 October 1975. This initial memo was the result of discussions over the course of the summer of 1975 between S. Fuller, D. Parnas, J. Shore, D. Siewiorek and W. Smith. Following discussion at the 1 and 2 October meeting a Selection Criteria subcommittee was formed to further develop specific criteria the CFA committee could use in selecting a computer architecture

	Organization
S. Fuller (Chairman)	NRL/CMU
R. Estell	FCDSSA, San Diego
L. Haynes	NSWC, White Oaks
N. Tinkelpaugh	NELC
D. Wise	USAMICOM

The Selection Criteria subcommittee met on 28 and 29 October 1975 and the results of our discussion were reported to the CFA committee at the 3 and 4 December 1975 meeting [Fuller et al., 1975]. In that report the original versions of the S, M, and R measures were presented. In the original definition of the M and R measures, they were in fact a set of measures, parameterized by the width of the bus. In other words, rather than just M, and M_1, M_8, M_{16},..., M_{64} were proposed where the subscript was the width of the bus between memory and the processor. The present definition of M is in fact M_4. The other bus widths were dropped because the variation of bus width added a dimension of complexity in the selection process that was felt to be of limited utility to the purposes of the CFA committee. Following the December 1975 CFA meeting a revised definition of the S, M, and R measures data was written and distributed to the CFA committee on 15 January 1976 [Burr, 1976]. This report still left a number of troublesome loose ends, particularly with respect to the R measure, and finally a memo on 24 May 1976 was written to clarify these definitions [Fuller, 1976]. Many people contributed ideas and suggestions to the final definition of the S, M, and R in addition to the original Selection Criteria committee. The most active contributors were W. Burr, D. Siewiorek, M. Barbacci, W. Gordon, and W. Smith. For the purposes of the CFA committee, the working definition of the S, M, and R measures are given in very concrete terms in Appendix D where these measures are specifically defined for each instruction, addressing mode, and instruction format for each of the three final candidate architectures. As should be clear from the definition of the R measure, and to some extent the M measure, there is a limited but nonnegligible degree of freedom in applying these measures to a specific architecture. Care was taken to be as consistent as possible in calculating the measures for the IBM S/370, PDP-11, and Interdata 8/32 and a single individual, W. Burr, examined (and did much of the computation) of the M and R measures for all three candidate architectures.

Both the 20 November 1975 and 15 January 1976 reports on the S, M, and R measures proposed a relative weighting technique similar to the method used with the quantitative criteria (see Volume II). However, at the third CFA meeting on 18-20 February 1976 this weighting scheme was rejected and replaced with the life-cycle cost analysis described in Volume VI. The S, M, and R values are now input as coefficients to terms estimating the memory size and processor speed of future tactical computers.

4. STATISTICAL DESIGN OF TEST PROGRAM ASSIGNMENTS*

Selection of statistical experimental designs to compare the candidate computer architectures involved several major considerations. These are common to all situations in which a statistical design plan must be chosen.

At an immediate practical level, an experimental design selected must be executable with the available resources, including time, funds, manpower, and available hardware and software. Moreover, the design must lead to experimental results which allow assessment of the main issues under study. That is, there must be a direct connection between the questions which can be answered from the data gathered and the questions formulated at the outset of the study. Finally, the experimental results must be capable of being analyzed by clearly understood statistical methods which do allow the assessment of the main issues of the study. These considerations influenced strongly the broad outlines of the designs chosen for comparison of the candidate architectures.

The test program phase of the computer family architecture evaluation process involved comparison of twelve test programs on three machines. Approximately sixteen programmers were available for the study and a complete factorial design would have required each programmer to write all of the test programs on each of the machines (for a total of 576 programs). This was clearly not feasible with the given time and resource constraints, and, consequently, a fractional design or several fractional designs had to be selected. Fractional factorial designs are discussed by [Davies, 1971], e.g. The fractional designs to be described below incorporate balance in the way test program, machine, and programmer combinations are assigned.

It was necessary to consider designs which required each programmer to write test programs for all three machines. Otherwise, comparisons among the machines could not be separated from comparisons among the programmers. A desirable design would have instructed each programmer to write a total of six or nine different test programs, one third of them on each of the three machines. For most of the programmers in the study time limitations precluded this type of design, and some compromise was required. The compromise design selected also had to allow for precise comparisons among the three competing architectures. A type of design that meets both of these objectives is the nested factorial [Anderson and McLean, 1974].

The test program part of the study actually involved the use of three separate experimental designs, henceforth referred to as Phase I, Phase II, and Phase III. Nested factorial designs were used for Phase I and Phase III; Phase II did not use a nested factorial design and will be discussed later in this section. Phase I was used to study test programs A through H, those deemed to be of primary interest. Phase III was used to study test programs I through L.

*Sections 4 and 5 describe the statistical design and analysis of the test program assignments and resulting data. Those readers who do not want to study the statistical considerations of the test program study may skip ahead to Section 6 to get a summary of the principal results of the analysis described in Sections 4 and 5.

The Phase I design is a pair of nested factorials. The plan of the design is shown in Figure 4-1. Each programmer thus writes each of two assigned test programs on all three machines. Moreover, each of the test programs is assigned to two different programmers. When this design was originally formulated, the plan included requiring each programmer to write his six test programs in a preassigned randomly selected order, so as to eliminate possible biases due to learning and gaining of experience during the course of completing the assignments. This procedure was discarded, however, when the programmers objected because of the varying availability of the three machines for debugging. Programmers were instructed to complete the assigned jobs in conformity with their typical practices and working habits with regard to order, consultation with other individuals, and other such considerations. Programmers in the study were not permitted to consult with each other, however, on any substantive matters of completing their designated assignments. All programmers were instructed to keep diaries of their work on the experiment.

As noted above in the discussion of the nested factorial design, the Phase I design was formulated with the goal of obtaining maximum possible information about differences between the competing architectures. With the given Phase I design, comparisons among the three architectures are theoretically not confounded by differences among either test programs or programmers. The eight test programs included in the Phase I design are those of main interest to the committee. The Phase I design was viewed as the most important of the three designs formulated. It focused attention on direct comparisons of the architectures, treated the test programs of primary interest, and called for the largest number of observations among the three chosen designs.

The design termed Phase III was formulated according to the same plan as was Phase I, except that four test programs and four programmers were utilized. The design layout is shown in Figure 4-2. The Phase III design contains half as many observations as the Phase I design and thus gives statistical results of less precision. The test programs in the Phase III design are of lesser interest to the committee than those in Phase I. The four programmers in Phase III are distinct from the eight in Phase I.

Together Phase I and Phase III designs provide a view of all three machines and the operation of all twelve test programs selected by the committee. A third experiment, labelled Phase II, was also planned. This was viewed as an auxiliary effort, designed to provide information additional to that given by Phase I and Phase III. Phase II was to be completed only if it was clear that the programmers assigned to it would not be needed to aid in the completion of Phase I and Phase III. The Phase II design called for three programmers to write nine different test programs, three on each of the three machines. The programmers assigned to Phase II were able to devote enough time to the test program study to permit use of a design which required them to write nine different programs. Six of the Phase I programs and three of the Phase III programs were selected for Phase II. Successful completion of Phase II was viewed at the outset as somewhat of a bonus. Some comparisons among programs not possible in Phase I and Phase III could be made, and the statistical results of Phase II could be compared to those of the other two experiments. The Phase II design was formulated as a one-third fraction of a 3^4 complete factorial design. The 3.4.3 plan in [Connor and Zelen, 1959] was used. This was made possible by dividing the factor test programs, which appears at nine levels, into two pseudofactors, each at three levels. The layout of the Phase II design is shown in Figure 4-3. One of the Phase II programmers also participated in the Phase I design. The only duplicate assignment, however, was test program G on the IBM S/370.

Programmer	1			2			3			4			5			6			7			8		
Machine	11	370	8/32	11	370	8/32	11	370	8/32	11	370	8/32	11	370	8/32	11	370	8/32	11	370	8/32	11	370	8/32
Test Program A	X	X	X													X	X	X						
B							X	X	X										X	X	X			
C				X	X	X																X	X	X
D										X	X	X	X	X	X									
E				X	X	X							X	X	X									
F										X	X	X				X	X	X						
G							X	X	X													X	X	X
H	X	X	X																X	X	X			

Figure 4-1. Phase I Design

Programmer	1			2			3			4		
Machine	11	370	8/32	11	370	8/32	11	370	8/32	11	370	8/32
Test Program I	X	X	X							X	X	X
J	X	X	X				X	X	X			
K				X	X	X				X	X	X
L				X	X	X	X	X	X			

Figure 4-2. Phase III Design

Programmer	1			2			3		
Machine	11	370	8/32	11	370	8/32	11	370	8/32
Test Program A		X		X					X
B	X					X		X	
E			X		X		X		
F	X					X		X	
G	X					X		X	
H			X		X		X		
J			X		X		X		
K		X		X					X
L		X		X					X

Figure 4-3. Phase II Design

5. ANALYSIS OF TEST PROGRAM RESULTS*

This section describes the experimental results and statistical analysis of the test program data. The outcomes of the statistical calculations are interpreted in light of the goals and purposes of the experiment. We shall first focus attention upon the Phase I experiment. The the Phase II experiment will be discussed. Some analysis combining data from Phase I and Phase III will then be presented. Finally, the Phase II experiment will be described.

a. PHASE I MODELS

As noted in Section 4, the Phase I experiment consists of a pair of nested factorial designs. A possible model for these nested factorial designs is

$$y_{ijk} = C + P_i + T_{ij} + M_k + PM_{ik} + TM_{ijk} + e_{ijk}, \qquad (5.1)$$
$$i = 1,2,3,4, \quad j = 1,2, \quad k = 1,2,3.$$

In this equation y_{ijk} is some response generated by the ith programmer writing the jth test program on the kth machine. Also,

C = constant, termed the grand mean

P_i = effect due to the ith programmer

T_{ij} = effect of the jth test program assigned to the ith programmer

M_k = effect of the kth machine

PM_{ik} = interaction between the ith programmmer and the kth machine

TM_{ijk} = interaction between the jth test program written by the ith programmer and the kth machine

e_{ijk} = a random error term, assumed to be normally distributed with mean 0 and variance not dependent on the values of i, j, and k.

Figure 4-1 indicates that programmers 1-4 are in one nested factorial design and programmers 5-8 are in the other. Data values from the Phase I experiment were analyzed using the analysis of variance (ANOVA), as applied to the nested factorial design. Some summary values resulting from the ANOVA calculations are displayed and discussed below.

*As mentioned in Section 4, readers may skip ahead to Section 6 to get a summary of the principal test program results.

The Phase I experiment may also be modelled in a manner different from that described above. In Phase I there are two factors at eight levels each, programmers and test programs, and one factor at three levels, machines. The two eight-level factors may each be replaced by three pseudofactors at two levels each. Consider the eight programmers, who constitute a single factor at eight levels. Now define three factors at two levels each. Label these factors A, B, and C, and code their levels as 0 and 1. Then the correspondence between the levels of the three pseudofactors and the levels of the original factor (programmers) is shown in Figure 5-1. We are concerned with a complete factorial experiment involving $3*2^6 = 192$ total observations. The actual Phase I experiment is a 1/4 fraction of this. A model may be fit using dummy variables to account for various effects and interactions.

b. TRANSFORMATION OF THE DATA

The discussion in this subsection is applicable to all three designs, Phase I, Phase II, and Phase III. Values of the S, M, and R measures for all the experiments are tabulated in Appendix E. Examination of the data given in Appendix E clearly shows there is wide variation in the data from one test program to another, e.g., especially for the M and R measures. Various statistical considerations suggest that some transformation of the raw data prior to analysis is desirable. A technical discussion of transformation of statistics is given by [Rao, Section 6g, 1973], who illustrates use of the methodology in various contexts.

In the CFA study the purpose of a transformation of the data is to stabilize variance, so that a model of the form (5.1) will hold. Specifically, the model (5.1) assumes that the variance of the error term e_{ijk} is constant, or independent of i, j, and k. Under this assumption inferences which follow from ANOVA calculations, as described below, are valid.

A variance stabilizing transformation is frequently suggested by consideration of the experimental situation and prior understanding of the variation to be expected in the data. For example, consider the M and R measures. Suppose some programmers each write two test programs and the average run time of the second one is k times the run time of the first. Then if the standard deviation of the M or R readings is V for the first test program, it can be expected to be proportional to kV for the second test program. In other words, the variability (standard deviation) in run times is directly proportional to the average run time. The accuracy of this conjecture will be tested in the analysis discussed in the next section, but clearly there is strong intuitive support for this assumption. Consider the Runge-Kutta test program. Its M and R measures are dominated by the computation of the inner loop performing the step-wise solution of the differential equation. Variations in M and R measures will be a result of alternative encodings of this inner loop. Average M and R measures will be doubled if the number of iterations requested is doubled. Moreover, doubling the number of iterations will also cause differences between the different Runge-Kutta programs to double. When the standard deviation of the test data is directly proportional to the mean, a logarithmic transformation will stabilize the variance, that is, remove the dependence of the variance on the size of the test program [Rao, Sec. 6g, 1973].

The model of (5.1) may be termed an additive model. That is, each observation is modelled as a sum of various effects and interactions, and the statistical error term is also involved additively.

Programmers	Pseudofactors A	B	C
1	0	0	0
2	0	0	1
3	0	1	0
4	0	1	1
5	1	0	0
6	1	0	1
7	1	1	0
8	1	1	1

Figure 5-1. Correspondence Between Levels of a Given Factor (Programmers) and Levels of Pseudofactors Used as Substitutes.

When a logarithmic transformation is used for the data, y_{ijk} in (5.1) becomes the logarithm of the response, such as the M or the R reading. In this case, a multiplicative model in fact underlies (5.1). We may write

$$z_{ijk} = \chi\pi_i \tau_{ij} \mu_k \pi\mu_{ik} \tau\mu_{ijk} \epsilon_{ijk}, \qquad (5.2)$$
$$i = 1,2,3,4 \quad j = 1,2, \quad k = 1,2,3.$$

The connection between (5.1) and (5.2) is

$$\ln z_{ijk} = y_{ijk}$$

$$\ln \chi = C,$$

$$\ln \pi_i = P_i,$$

$$\ln \tau_{ij} = T_{ij},$$

$$\ln \mu_k = M_k,$$

$$\ln \pi\mu_{ik} = PM_{ik},$$

$$\ln \tau\mu_{ijk} = TM_{ijk},$$

$$\ln \epsilon_{ijk} = e_{ijk}.$$

Thus, use of the logarithmic transformation on both sides of (5.2) yields (5.1), and the multiplicative model (5.2) may be viewed as the meaningful basic underlying model. As noted above, this situation seems to arise naturally in consideration of the M and R measures.

Similar considerations for the S measure suggest a square root transformation is appropriate to stabilize its variance. This transformation arises because the variance, rather than the standard deviation, of the S measure of the second test program in the above discussion can be expected to be proportional to kV. Use of the square root transformation would imply use of the model in (5.1) with y_{ijk} denoting the square root of the measured S value.

To summarize this discussion, we note that the purpose of a variance stabilizing transformation in the CFA study is twofold. First, a transformation may have underlying it a meaningful model for the data, such as the multiplicative model (5.2). And second, stabilization is necessary to justify the use of ANOVA techniques to analyze the data. It should be noted that the square root and logarithmic transformations are only two of a large number of transformations. A family of practical transforms takes a response z and transforms it according to z^{α} for any $\alpha > 0$. With an appropriate interpretation, the logarithmic transformations correspond to the limiting value $\alpha = 0$. This family of power transformation is discussed in detail by [Box and Cox, 1964]. In the present study only the square root and logarithmic transformations have been used. As discussed above, the former transformation appears to be appropriate for the S measure and the latter for the M and R measures. Results of statistical analysis for both transformations on all three measures will in fact by presented below.

c. STATISTICAL ANALYSIS OF PHASE I DATA

Analysis of variance calculations were performed on both halves of the Phase I experiment for $\sqrt{S}$, $\sqrt{M}$, $\sqrt{R}$, ln S, ln M, and ln R measures. Each analysis on a half of the Phase I experiment involved 24 data values. In each analysis the sample variance of the 24 values was decomposed into sums of squares attributable to variations among programmers, test programs, machines, programmer-machine interactions, and test program-machine interactions. The proportions of the total variance due to the various sums of squares are summarized in Table 5-1.

These analysis of variance calculations indicate that test program and programmer variations account for most of the variation in the data in the case of the M and R measures, and that machine differences are extremely small on a relative scale. Machine differences are noticeable for the S measure. More detailed and precise comparisons of the three machines are presented below.

Table 5-1 shows that in the case of the S measure there are some differences between the two sets of programmers (1-4 vs. 5-8). Each half of the Phase I experiment is rather small, involving only four programmers and 24 observations. The outcomes for the S measure suggest that a larger sample of programmers may be necessary to obtain results that representative of the population of professional programmers.

At the end of Section 5.1 an alternative formulation for the model of the Phase I experiment was discussed. This formulation viewed the experiment as a fractional factorial design and replaced each of two factors by a set of pseudofactors. Using dummy variables, we fit models to sets of Phase I data with $\sqrt{S}$, $\sqrt{M}$, $\sqrt{R}$, ln S, ln M, and ln R as the responses. In each model 24 parameters were fit, leaving 24 degrees of freedom to measure experimental error. Estimates of the variance of the error term in the model (5.1) are shown for each type of response in Table 5-2. The estimates of variance shown in Table 5-2 may be used to construct confidence intervals for comparisons among the three architectures. As in (5.1), let M_k. denote the differential effect due to the kth machine. We establish the following convention:

M_1 = effect of the PDP-11

M_2 = effect of the IBM S/370

M_3 = effect of the Interdata 8/32.

Table 5-3 shows estimates of various machine comparisons for the Phase I data. A 95% confidence interval is quoted below each estimate. The confidence intervals which do not cover the value 0 correspond to comparisons significant at level .05 (=1-.95). Thus at level .05 the Interdata 8/32 is superior to the IBM S/370 on all measures. For the M_2-M_1 comparison the PDP-11 is adjudged superior at level .05 on four of the measures and barely misses being superior when $\sqrt{S}$ and $\sqrt{M}$ responses are considered. Moreover, the IBM S/370 is inferior to the average performance of the other two machines on all measures. The estimates of the comparisons M_3-M_1 and $1/2(M_1+M_3)$-M_2 are statistically independent, and no other pairs are independent. It is worth noting that these comparisons among the competing architectures are based upon consideration of test programs A through H only. It is reasonable, however, to view the eight programmers in Phase I as representative of a large population of programmers.

Measure		√S	√S	√M	√M	√P	√R	ln S	ln S	ln M	ln M	ln R	ln R
Programmers		1-4	5-8	1-4	5-8	1-4	5-8	1-4	5-8	1-4	5-8	1-4	5-8
Sum of square	Degrees of freedom												
Programmers	3	.078	.362	.377	.391	.381	.396	.088	.423	.337	.461	.327	.471
Test Programs	4	.294	.342	.613	.594	.616	.597	.309	.218	.611	.489	.621	.473
Machines	2	.133	.064	.002	.003	.001	.002	.144	.089	.014	.020	.015	.023
Programmers X Machines	6	.180	.105	.004	.004	.001	.001	.160	.123	.015	.013	.013	.014
Test Programs X Machines	8	.314	.126	.004	.007	.001	.003	.300	.147	.022	.018	.024	.018

Table 5-1. Phase I ANOVA Calculations
Proportion of Variance Attributable to Each Sum of Sources

Response	Estimate of σ^2
$\sqrt{S}$	18.175
$\sqrt{M}$	272.546
$\sqrt{R}$	577.863
ln S	.398
ln M	.377
ln R	.400

Table 5-2. Estimates σ^2 of the Variance of the Error Component in Model (5.1), Phase I Data, 24 Degrees of Freedom

Comparison of Machines	Measure					
	S	M	R	ln S	ln M	ln R
$M_3 - M_1$	-.586	-4.550	-8.926	-.057	.018	.012
	(-3.696,2.524)	(-16.598,7.498)	(-24.468,10.616)	(-.517,.403)	(-.430,.466)	(-.449,.474)
$M_3 - M_2$	-3.535	-14.855	-28.216	-.559	-.655	-.717
	(-6.645,-.425)	(-26.903,-2.807)	(-45.758,-10.674)	(-1.019,-.099)	(-1.103,-.207)	(-1.178,-.255)
$M_2 - M_1$	2.949	10.305	21.290	.502	.673	.729
	(-.161,6.059)	(-1.743,22.353)	(3.748,38.832)	(.041,.962)	(.225,1.121)	(.267,1.191)
$\frac{1}{2}(M_1+M_3)-M_2$	-3.242	-12.580	-24.753	-.531	-.664	-.723
	(-5.936,-.548)	(-23.014,-2.147)	(-39.944,-9.562)	(-.929,-.132)	(-1.052,-.276)	(-1.122,-.323)

model (5.1):

M_1: effect of PDP-11

M_2: effect of IBM S/370

M_3: effect of Interdata 8/32

Table 5-3. Estimates of Machine Comparisons and 95% Confidence Intervals, Phase I

Table 5-4 displays estimates of the effects M_k and μ_k for the various measures. The latter are obtained by exponentiating the estimates of M_k and are appropriate for the logarithmic models only. Since the effects noted in Table 5-4 are differential values, a value of 0 is neutral for M_k and a value of 1 is neutral for μ_k. The figures in Table 5-4 are consistent for the different measures and transformations. The IBM S/370 is noticeably worse than the other two machines. For the responses $\sqrt{S}$, $\sqrt{M}$, and $\sqrt{R}$, the Interdata 8/32 appears to be modestly better than the PDP-11.

One may interpret the last three lines of Table 5-4 in the following way. The IBM S/370 requires 142.5% as much space to represent test programs A through H as the average of the three machines, while the PDP-11 and the Interdata 8/32 require 86.2% and 81.5% as much space, respectively. Corresponding statements may be given for execution times as reflected by the ln M and ln R measures.

Measure	$\sqrt{S}$	$\sqrt{M}$	$\sqrt{R}$	ln S	ln M	ln R
Comparison of Machines						
M_1	-.788	-1.918	-4.788	-.148	-.230	-.247
M_2	2.161	8.387	16.502	.354	.443	.482
M_3	-1.374	-6.468	-11.714	-.205	-.212	-.235
μ_1				.862	.795	.781
μ_2				1.425	1.557	1.619
μ_3				.815	.809	.791

M_1, μ_1: effects for PDP-11

M_2, μ_2: effects for IBM S/370

M_3, μ_3: effects for Interdata 8/32

Table 5-4. Estimates of Machine Effects in Models (5.1) and (5.2), Phase I

d. PHASE III MODELS AND RESULTS

The models for Phase III experiments are the same as in (5.1) and (5.2), except that the subscript i assumes the values 1 and 2 only. The layout of the Phase III designs is illustrated in Figure 4-2.

Table 5-5 gives a summary of analysis of variance calculations for the Phase III data. It is comparable to Table 5-1.

Estimates of the variance of the error term in the Phase III version of model (5.1) are shown in Table 5-6, which is the analog of Table 5-2. The estimates are based on eight degrees of freedom. That is, models were fit to sets of Phase III data with $\sqrt{S}$, $\sqrt{M}$, $\sqrt{R}$, ln S, ln M, and ln R as the responses. In each model 24 data values were employed, and dummy variables were used to fit 16 parameters. It should be noted that the Phase III estimate of σ^2 have less precision than those of Phase I -- they are based on eight degrees of freedom rather than 24. The corresponding figures in Tables 5-2 and 5-6 are comparable, though, with several of the pairs very close.

Table 5-7 is the analog of Table 5-3, and Table 5-8 the analog of Table 5-4. None of the confidence intervals shown in Table 5-7 fails to cover the value 0. However, it is apparent that the PDP-11 performed noticeably worse than the other two machines in Phase III. Also, there is very little difference between the IBM S/370 and the Interdata 8/32 in Phase III.

The relatively poor performance of the PDP-11 in Phase III appears to be due to its inability to handle test program I, quicksort. Certainly part of the explanation for the poor performance of the IBM S/370 in Phase I can be attributed to test program A, I/O kernel with four priority levels. In the next section results from Phase I and Phase III are combined to produce overall estimates of machine effects and overall comparisons of the machines.

e. COMBINATION OF PHASE I AND PHASE III RESULTS

Let θ_I denote an estimate of a machine effect or comparison, such as M_1 or M_3-M_1, in Phase I. Let θ_{III} denote the estimate of the same effect or comparison in Phase III. In the previous two sections such estimates were given, as well as some confidence intervals. The purpose of this section is to present estimates of the form

$$\alpha\theta_I + (1-\alpha)\theta_{III},$$

where α is chosen to minimize the variance of the resulting linear combination and $0 < \alpha < 1$. Table 5-9 shows estimates of machine comparisons and 95% confidence intervals. The value of α for each column in the table is given along the top border. In all columns but the fourth more weight is given to the Phase I data. Table 5-10 gives estimates of machine effects with Phase I and Phase III data combined.

Five of the confidence intervals for M_3-M_2 in Table 5-9 fail to cover the value zero, and the sixth one (for M) almost fails to cover. Thus, the evidence suggests that the Interdata 8/32 performs better than the IBM S/370 on all three measures, S, M, and R. Also, the IBM S/370 tends to be worse than the average of the other two machines.

Measure		√S	√S	√M	√M	√R	√R	ln S	ln S	ln M	ln M	ln R	ln R
Programmers		1-2	3-4	1-2	3-4	1-2	3-4	1-2	3-4	1-2	3-4	1-2	3-4
Sum of squares	Degrees of freedom												
Programmers	1	.112	.096	.083	.383	.103	.505	.090	.069	.030	.416	.061	.567
Test Programs	2	.391	.503	.419	.371	.435	.340	.463	.573	.632	.369	.659	.288
Machines	2	.081	.075	.129	.030	.116	.026	.062	.068	.077	.014	.061	.019
Programmers X Machines	2	.037	.147	.097	.074	.095	.029	.007	.136	.036	.070	.030	.028
Test Programs X Machines	4	.379	.179	.273	.142	.252	.100	.377	.154	.224	.131	.189	.098

Table 5-5. Phase III ANOVA Calculations Proportions of Variance Attributable to Each Sum of Squares

Response	Estimate of σ^2
$\sqrt{S}$	18.606
$\sqrt{M}$	245.688
$\sqrt{R}$	1079.745
ln S	.174
ln M	.374
ln R	.308

Table 5-6. Estimate of σ^2 of the Variance of the Error Component in Model (5.1), Phase III Data, Eight Degrees of Freedom.

Comparison of Machines	Measure					
	$\sqrt{S}$	$\sqrt{N}$	$\sqrt{R}$	ln S	ln N	ln R
$M_3 - M_1$	-3.806	-10.457	-25.229	-.307	-.295	-.348
	(-8.780,1.168)	(-28.529,7.615)	(-63.117,12.659)	(-.787,.173)	(-1.000,.410)	(-.988,.291)
$M_3 - M_2$	-1.585	2.963	1.051	-.216	-.099	-.027
	(-6.559,3.389)	(-15.109,21.035)	(-36.837,37.939)	(-.696,.264)	(-.606,.804)	(-.666,.613)
$M_2 - M_1$	-2.221	-13.420	-26.280	-.091	-.394	-.321
	(-7.195,2.753)	(-31.492,4.652)	(-64.168,11.608)	(-.571,.389)	(-1.099,.311)	(-.960,.318)
$\frac{1}{2}(M_1+M_3)-M_2$	.318	8.192	13.666	-.063	.247	.147
	(-3.990,4.626)	(-7.459,23.842)	(-19.147,46.478)	(-.478,.353)	(-.364,.858)	(-.407,.701)

model (5.1): M_1: effect of PDP-11

M_2: effect of IBM S/370

M_3: effect of Interdata 8/32

Table 5-7. Estimates of Machine Comparisons and 95% Confidence Intervals, Phase III

Measure	$\sqrt{S}$	$\sqrt{M}$	$\sqrt{R}$	ln S	ln M	ln R
Comparison of Machines						
M_1	2.009	7.959	17.169	.133	.229	.223
M_2	-.212	-5.461	-9.111	.042	-.165	-.098
M_3	-1.797	-2.498	-8.060	-.174	-.066	-.125
1				1.142	1.257	1.250
2				1.083	.848	.907
3				.840	.936	.882

μ_1, α_1: effects for PDP-11

M_2, α_2: effects for IBM S/370

M_3, α_3: effects for Interdata 8/32

Table 5-8. Estimates of Machine Effects in Models (5.1) and (5.2), Phase III

Comparison of Machines	Measure: $\sqrt{S}$ α=.67	$\sqrt{M}$ α=.64	$\sqrt{R}$ α=.79	ln S α=.47	ln M α=.66	ln R α=.61
$M_3 - M_1$	-1.649	-6.676	-10.770	-.190	-.088	-.128
	(-4.119,.821)	(-16.039,2.685)	(-25.868,4.323)	(-.494,.114)	(-.442,.266)	(-.517,.261)
$M_3 - M_2$	-2.892	-8.441	-22.070	-.377	-.39_	-.448
	(-5.362,-.422)	(-17.803,.921)	(-37.168,-6.972)	(-.681,-.073)	(-.753,-.045)	(-.837,-.059)
$M_2 - M_1$	1.243	1.764	11.300	.188	.310	.320
	(-1.227,3.713)	(-7.598,11.126)	(-3.798,26.398)	(-.116,.492)	(-.044,.664)	(-.069,.708)
$\frac{1}{2}(M_1+M_3)-M_2$	-2.067	-5.102	-16.685	-.283	-.354	-.384
	(-4.207,.073)	(-13.210,3.006)	(-29.761,-3.609)	(-.547,-.019)	(-.661,-.047)	(-.721,-.047)

M_1: effect of PDP-11

M_2: effect of IBM S/370

M_3: effect of Interdata 8/32

Table 5-9. Estimates of Machine Comparisons and 95% Confidence Intervals, Phase I and Phase III Data Combined

The estimates of μ_k in Table 5-10 provide a summary of the Phase I and Phase III data. The IBM S/370 requires 120.8% as much storage as the average of all three machines for the 12 test programs studied. According to the M measure estimate, the IBM S/370 requires 126.6% as much time to "execute" the test programs as the average of the three machines. The other figures in the lower part of Table 5-10 are interpreted similarly.

Measure	√S	√M	√R	ln S	ln M	ln R
Comparison of Machines	α=.67	α=.64	α=.79	α=.47	α=.66	α=.61
M_1	.135	1.638	-.177	.001	.075	.064
M_2	1.378	3.402	11.123	.189	.236	.256
M_3	-1.514	-5.039	-10.947	-.189	-.163	-.192
μ_1				1.001	.928	.938
μ_2				1.208	1.266	1.292
μ_3				.828	.850	.825

M_1, μ_1: effects for PDP-11

M_2, μ_2: effects for IBM S/370

M_3, μ_3: effects for Interdata 8/32

Table 5-10. Estimates of Machine Effects in Models (5.1) and (5.2), Phase I and Phase III Data Combined

f. PHASE II MODELS AND RESULTS

Analysis of variance calculations were performed on data arising from the Phase II design. Some of the results for responses √S, ln R, and ln M are summarized in Table 5-11. This table indicates the proportions of the total variance attributable to various sums of squares. The corresponding degrees of freedom are also indicated. The statistical analysis was performed by utilizing the theory associated with a 1/3 fraction of a 3^4 design. The variance was split into sums of squares each with two degrees of freedom. Since two of the factors in the design were in fact pseudofactors at three levels each to account for the nine test programs, several sets of sums of squares were combined. There is some aliasing in the design involving second-order interactions.

Measure		√S	ln M	ln R
Sum of squares	Degrees of freedom			
Programmers	2	.027	.018	.026
Test Programs	8	.623	.653	.660
Machines	2	.132	.076	.068
Programmers X Machines	2	.039	.053	.047
Test Programs X Machines	8	.132	.124	.121
Test Programs X Programmers	4	.047	.076	.078

Table 5-11. Phase II ANOVA Calculations Proportions of Variance Attributable to Each Sum of Squares

Estimates of differential effects in a model comparable to (5.1) for the three machines can also be given. For the √S measure they are -.952 for the PDP-11, 1.605 for the IBM S/370, and -.653 for the Interdata 8/32. For the ln M measure the values are -.691, .508, and .183 for the machines quoted in the same order, and the figures are -.602, .538, and .123 for the ln R measure. Thus, the experimental results tend to rank the machines with the PDP-11 first by a substantial margin, and the Interdata 8/32 ranks second. It should be noted that test program A was included in the Phase II design, and test programs D and I were not.

g. ANALYSIS OF PHASE I AND PHASE III USING ONLY THE "BEST PROGRAMS"

Each test program in the Phase I and Phase III designs was written by two different programmers. Moreover, the models (5.1) and (5.2) for these designs involved three factors, programmers, test programs, and machines. In this section we describe analysis of the data when the factor representing the programmers is eliminated from the designs. This is accomplished by selecting the smaller of the two S, M, or R readings when a specified test program is written for a specified machine. The interpretation of this approach to the analysis is that the "best" programme is being selected in each case. It is worth noting, however, that "best" is merely among two persons for the data in Phase I and Phase III.

When the designs are reduced to consideration of two factors in the manner described above, the model (5.1) is replaced by

$$y_{ij} = C + T_i + M_j + TM_{ij} + e_{ij}, \qquad (5.3)$$

where

y_{ij} = response when test program i is written for machine j,

T_i = effect of test program i,

M_j = effect of machine j,

TM_{ij} = interaction between test program i and machine j,

e_{ij} = a random error term, assumed to be normally distributed with mean 0 and constant variance σ^2.

The design (5.3) is a two-way layout (see [Anderson and McLean, 1974], e.g.). We present below the usual analysis of variance table for $\sqrt{S}$, ln M, and ln R responses for the Phase I and Phase III experiments.

First consider the $\sqrt{S}$ response for the Phase I design. Table 5-12 displays the ANOVA calculations. The sum of squares for machines has been decomposed into two parts, one involving the comparison M_1-M_3, and the other the comparison 1/2 (M_1+M_3)-M_2. The interaction/error sum of squares has been similarly decomposed. Details of this decomposition are presented by [Anderson and McLean, 1974], e.g. As above, the correspondence is M_1: PDP-11, M_2: IBM S/370, M_3: Interdata 8/32.

Source	Sum of squares	Degrees of freedom	Mean Square
Test Programs	274.387	7	39.198
Machines			
M_1-M_3	.361	1	.361
$\frac{1}{2}(M_1+M_3)-M_2$	37.989	1	37.989
subtotal	38.350	2	
Interaction/error			
(M_1-M_3)Xprograms	10.925	7	1.561
$[\frac{1}{2}(M_1+M_3)-M_2]$Xprograms	113.134	7	16.162
subtotal	124.059	14	
Total	436.796	23	

Table 5-12. ANOVA for $\sqrt{S}$, Phase I, Model (5.3)

Estimates of the differential effects M_1, M_2, and M_3 in (5.3) are -.739, 1.779, and -1.040, respectively.

Tables 5-13 and 5-14 present the same calculations for the ln M and ln R responses in Phase I. For ln M estimates of M_1, M_2, and M_3 are -.186, .401, and -.215, respectively. For ln R they are -.196, .424, and -.228, respectively.

Source	Sum of squares	Degrees of freedom	Mean Square
Test Programs	128.981	7	18.426
Machines			
M_1-M_3	.003	1	.003
$\frac{1}{2}(M_1+M_3)-M_2$	1.932	1	1.932
subtotal	1.935	2	
Interaction/error			
(M_1-M_3)Xprograms	.291	7	.042
$[\frac{1}{2}(M_1+M_3)-M_2]$Xprograms	3.701	7	.529
subtotal	3.992	14	
Total	134.908	23	

Table 5-13. ANOVA for ln M, Phase I, Model (5.3)

Source	Sum of squares	Degrees of freedom	Mean Square
Test Programs	136.962	7	19.965
Machines			
M_1-M_3	.004	1	.004
$\frac{1}{2}(M_1+M_3)-M_2$	2.161	1	2.161
subtotal	2.164	2	
Interaction/error			
(M_1-M_3)Xprograms	.492	7	.070
$[\frac{1}{2}(M_1+M_3)-M_2]$Xprograms	4.124	7	.589
subtotal	4.615	14	
Total	146.742	23	

Table 5-14. ANOVA for ln R, Phase I, Model (5.3)

The results of this "best" programmer analysis of the Phase I data are consistent with other results previously described. Variability in the data may be attributed primarily to differences among test programs, rather than differences among machines, and this is especially so for the M and R measures. Moreover, differences among the three machines are largely described by noting that the performances of the PDP-11 and Interdata 8/32 are comparable, and that the IBM S/370 is measurably worse than the other two. It is interesting to note that the decomposition of the interaction/error sum of squares shown in Tables 5-12-14 reflects this assessment of the three machines.

The comparisons described above for the three machines were obtained for an experiment involving test programs A-H. In the Phase III experiment test programs I-L were utilized and the discussion in Section 5.4 indicated a different ranking of the three machines was obtained. Similar results occur for the "best" programmer analysis. Table 5-15 shows analysis of variance calculations for the response $\sqrt{S}$. Estimates of the effects M_1, M_2, and M_3 in (5.3) are 1.403, -.162, and -1.241, respectively. The decompositions in Table 5-15 use M_2-M_3 and $1/2\ (M_2+M_3)-M_1$, which differ from the comparisons in Tables 5-12-14, it should be noted.

Source	Sum of squares	Degrees of freedom	Mean Square
Test Programs	161.399	3	53.800
Machines			
M_2-M_3	2.327	1	2.327
$\frac{1}{2}(M_2+M_3)-M_1$	11.812	1	11.812
subtotal	14.139	2	
Interaction/error			
(M_2-M_3)Xprograms	13.114	3	4.371
$[\frac{1}{2}(M_2+M_3)-M_1]$Xprograms	77.358	3	25.786
subtotal	90.471	6	
Total	266.009	9	

Table 5-15. ANOVA for $\sqrt{S}$, Phase III, Model (5.3)

The discussion in this section of analysis of the "best" programmer model (5.3) may be accurately summarized by stating that the results are very similar to those obtained in Sections 5.3 and 5.4.

h. THE PROGRAMMERS' LOGS

In the original memo to programmers on 8 April 1976 the programmers were asked to keep a diary of how their time on the project was spent. At the time the request was made, it was expected that the primary use of the diaries would be in attempting to understand anomalies in the statistical analysis. Though the memo requested a resolution of hours or half-days, and asked that the time be delineated into several categories, only seven of the programmers kept detailed enough records to provide any quantitative information about how programmer time was spent. Even among these seven, the time scale recorded, and the separation of protions of the task, varied a great deal. An eighth programmer recorded information on an hourly basis, but failed to make any distinction among the different phases of the task.

The availability of facilities had a large effect on the debugging time recorded. The PDP-11 programs were debugged interactively with a symbolic debugging package*, on a machine where a simple debugging package is part of the microcode.** The IBM S/370 programs were debugged on a batch system, where the debugging tools were core dumps and the trace of parameters values provided by the driver. For the Interdata 8/32 programs, almost no debugging time was recorded because most of the debugging was done by one of the programmers who made periodic weekend journeys to the only available Interdata site (Oceanport, N.J.). He recorded only the time spent debugging his own programs. When a programmer recorded an hour debugging on the S/370, it was not clear how much of that time was spent examining the listing and how much was spent waiting for batch runs.

Even with the seven programmers mentioned, there was often not enough separation of information. Sometimes a programmer would record "four hours spent studying all three hardware manuals"; it was impossible to tell in such instances how much time was spent studying any one machine.

With these caveats in mind, we present the following tables summarizing the programmer logs. The numbers in parentheses represent the number of programmers on which the times are based.

* Dynamic Debugging Tool, DDT
**The Digital Equipment Corporation LSI-11

Total Time Spent, Per Machine Per Programmer

	PDP-11	IBM S/370	Interdata 8/32
Study	18 (6)	22 (5)	33 (6)
Coding	48 (7)	56 (7)	46.5 (7)
Debugging	48.5 (7)	30 (7)	17.5 (5)

Total Time Spent, Per Problem Per Programmer

	Study	Coding	Debugging
A		2 (2)	.5 (1)
B	.5 (1)	20 (3)	
C		16 (1)	4 (1)
D			
E		13 (3)	9.5 (3)
F		3.5 (1)	2 (1)
G	6.5 (2)	19.5 (3)	12 (1)
H		4 (2)	2 (2)
I		4.5 (1)	14 (1) 33 (1)
J	5 (2)	11.5 (3)	3 (3)
K	2 (1)	24.5 (4)	19 (4)
L	6.5 (3)	20 (4)	5.5 (3)

i. CHRONOLOGY OF TEST PROGRAM ASSIGNMENT AND ANALYSIS

The statistical approach to the design of the test program assignments was developed by P. Snaman and S. H. Fuller prior to the third CFA committee meeting on 19-20 February 1976. The approach was explained at the meeting, a description of the methodology was distributed [Fuller and Snaman, 1976], and the CFA committee agreed to use this approach in the test program assignments. Volunteer programmers were requested at the February CFA meeting and assignments were sent out to eight programmers in various Army and Navy laboratories on 8 April 1976. During the February CFA meeting, it became clear that we could not complete the test program study without programmers in addition to the eight Army/Navy volunteers. Therefore, a proposal was submitted by Carnegie-Mellon University (CMU) to the Army Research Office, Durham, N.C., on 7 April 1976 for support of graduate students at CMU to write test programs. The proposal was accepted and CMU was awarded a grant (DAAG29-76-G-0299) to complete the test program study. Nine additional programmers from CMU were assigned test programs and by mid-July, 1976, all but three of

the 99 test programs in Phases I, II, and III were written, debugged, and S, M, and R measurements completed. Fortunately, a few "auxiliary" test programs in addition to the basic 99 were written and we were able to estimate values for the three missing data points. (The missing data points and the estimated values used by the CFA committee are shown in Appendix E. When these missing data points did become available no significant changes in the results were found.)

The group of graduate students at CMU proved crucial to the completion of the full set of assigned test programs. We gratefully acknowledge their participation. Three of these students, Leland Szewcrenko, George Mathew, and Narindra Jain have been particularly helpful through their continued effort on behalf of this project.

The results of the test program study were input to the life cycle cost models on 23 July 1976 and the entire CFA committee was given a summary of the results at the 24-26 August 1976 meeting [Fuller, Burr, and Shaman, 1976].

6. Summary

This volume of the Computer Family Architecture (CFA) Selection Committee Final Report has described how the test program phase of the CFA study was developed, what methodologies have been used, and what were the results of the study.

Section 2 described the rationale leading up to the selection of the 12 test programs used in the study.

a. I/O Kernel, Four Priority Levels

b. I/O Kernel, FIFO Processing

c. I/O Device Handler

d. Large Fast Fourier Transform

e. Character Search

f. Bit Test, Set Reset

g. Runge-Kutta Integration

h. Linked List Insertion

i. Quicksort

j. ASCII to Floating Point Conversion

k. Virtual Memory Space Exchange

The full specification of these test programs is given in Appendix A. The test programs were meant to span the range of application most important to DoD.

Section 3 of this volume defined the three measures of performance used to evaluate the candidate computer architectures on each test program:

S: Number of bytes used to represent a test program

M: Number of bytes transferred between primary memory and the processor during execution of the test program

R: Number of bytes transferred among internal registers of the processor during execution of the test program

In Sections 4 and 5 statistical results of the test program measurements are discussed. Section 4 deals with general and specific design considerations and statistical details presented in Section 5.

The test program study involved three different parts. These are labeled Phase I, Phase II, and Phase III. In Phase I eight programmers each wrote two test programs on each of the three machines. The programs in Phase I were A through H. In Phase III four programmers each wrote two test programs on each machine. The test programs in Phase III were I through L. The Phase II design

was implemented to attempt to corroborate information obtained from the other two designs. Three programmers each wrote nine different test programs, three on each of the machines. The test programs in Phase II were A, B, E, F, G, H, J, K, and L.

The principal results of the test program study that were passed on to the life-cycle cost models (see Volume VI) was the composite performance of the candidate architectures on the set of 12 test programs in Phases I and III. An Analysis of Variance (ANOVA) procedure was used to determine the overall relative performance of the three candidate machines. Unity indicates average performance and the lower the score on any of the measures, the better the machine handled the set of test programs.

Architecture	S	M	R
PDP-11	1.00	0.93	0.94
IBM S/370	1.21	1.27	1.29
Interdata 8/32	0.83	0.85	0.83

Table 6.1. Relative Performance of Candidate Architectures

In other words, our test program results indicate that the IBM S/370 needs 46% more memory than the Interdata 8/32 to represent the set of test programs (or 21% more than the average of the three architectures) and the PDP-11 is essentially average in its use of memory. Similarly, the PDP-11's ability to "execute" the test programs ranges between 93% and 94% of the average execution time based on the M and R measure, respectively.

Considering the test program results in a little more detail, in Phase I the data revealed the IBM S/370 to be significantly worse than the other two machines on S, M, and R measures at a confidence level of 95%. Moreover, the overall performance of the PDP-11 was virtually identical to that of the Interdata 8/32. Some part of the poor performance of the IBM S/370 can be traced to test program A (the priority I/O kernel).

In Phase III alone, none of the comparisons among the three machines was significant at a confidence level of 95%. The PDP-11 was noticeably the worst of the three machines on all three measures. The IBM 370 dominated the Interdata 8/32 with regard to the M measure, the Interdata was better for the S measure, and there was little difference between the two for the R measure. The relatively poor performance of the PDP-11 appeared to be due to test program I, a quicksort program working with a list much larger than the virtual address space of the PDP-11 (64K bytes).

Statistical results from Phases I and III were combined. In this analysis the ranking of the three machines from best to worst on all three measures was: Interdata 8/32, PDP-11, and IBM 370. The PDP-11 was much closer to the Interdata 8/32 than the IBM 370 was to the PDP-11. The average performance for the three machines in Phases I and III is given above in Table 6.1. Figure 6.1 shows the 95% confidence intervals that surround comparisons of these average values.

The outcome of Phase II largely corroborates the results of the other two experiments. The ranking of three machines, from best to worst is: PDP-11, Interdata 8/32, IBM 370. This ranking prevails for all three measures, S, M, and R. For the M and R measures the PDP-11 is substantially better than the

Interdata 8/32. For the S measure, however, the PDP-11 and Interdata 8/32 are very close to each other in performance. It is important to recall that Phase II includes test program A, for which the IBM 370 performs relatively poorly, and does not include test programs D and I, which are relatively difficult to implement on the PDP-11 because they have very large data structures.

The experimental designs executed in the test program phase of the computer family architecture evaluation process permitted a quantitative comparison of the three machines in the study. The experiments were not large-scale, and the results need to be interpreted with some caution. In particular, only a very small sample of programmers was used in each phase, especially in II and III. Future studies of the same size and scope as the present one seem from the available evidence to be worthwhile undertakings. They could be used to attempt to corroborate the present results. Larger scale experiments would be much more informative. (Though these are likely to require substantially more funds, however.) The main statistical issues appear to be the need to use a representative sample of programmers and to obtain estimates of parameters with high precision.

Another important point emerging from this study is that there is a significant interaction between the architecture under consideration and the test program being written. This in fact was known a priori to be an important factor. Its influence is clear in the contrast of Phase I, Phase II, and Phase III results.

REFERENCES

1. Anderson, V. L. and McLean, R. A., Design of Experiments a Realistic Approach, Marcel Dekker, Inc., New York, 1974.

2. Bell, C. G. and A. Newell, Computer Structures: Readings and Examples, McGraw Hill, New York, 1971.

3. Bonwell, N. (editor), Benchmarking: Computer Evaluation and Measurement, John Wiley & Sons, New York, 1975.

4. Box, G. E. P. and Cox, D. R., An Analysis of Transformations, The Journal of the Royal Statistical Society, Series B, Vol. 26 (1964), 211-252.

5. Brent, R. P., "On the precision attainable with various floating point number systems," IEEE Trans. on Computers C-22, 6 (June 1973), 601-607.

6. Buchholz, W. "A synthetic job for measuring system performance," IBM Systems Journal 8 (1969), 309-318.

7. Burr, W. E., "Computer Architecture Test Program Subcommittee Report," CFA memorandum, 15 January 1976.

8. Computer Review, GML Corporation, Lexington, Mass., 1976.

9. Conner, W. S. and Zelen, M., Fractional Factorial Experiment Designs for Factors at Three Levels, National Bureau of Standards, Applied Mathematics Series 54, 1959.

10. Davies, O. O. (ed.), Design and Analysis of Industrial Experiments, 2nd ed., Oliver and Boyd, Edinburgh, 1971.

11. Fuller, S. H., "Further comments and revisions to the S, M, and R Architecture measures," CFA memorandum, 24 May 1976.

12. Fuller, S. H., R. Estell, L. Haynes, N. Tinklepaugh, and D. Wise, "Selection Criteria for the Army/Navy Computer Family Architecture," CFA memorandum, 20 November 1975. Distributed at the 1-2 December 1975 CFA Selection Committee meeting.

13. Fuller, S. H., P. Shaman, and W. E. Burr, "Results of the CFA performance evaluation of the IBM S/370, PDP 11, and Interdata 8/32 computer architectures," CFA memorandum, 19 August 1976. Distributed at the 24-26 August 1976 CFA Selection Committee meeting.

14. Fuller, S. H. and P. Shaman, "Assignment of test programs to programmers," CFA memorandum, 16 February 1976. Distributed at 18-20 February 1976 CFA Selection Committee meeting.

15. Fuller, S. H. and W. R. Smith, "Selection Criteria for the Army/Navy Computer Family Architecture," CFA memorandum, 29 September 1975. Distributed at the 1-2 October 1975 CFA Selection Committee meeting.

16. IBM System/370 Principles of Operation, IBM Publication GA22-7000-3, White Plains, NY, 1973.

17. Lucas, H. C., "Performance evaluation and monitoring," ACM Computing Surveys 3, 3, (1971) pp. 79-91.

18. Rao, C. R., Linear Statistical Inference and Its Applications, 2nd ed., John Wiley & Sons, New York, 1973.

19. Stone, H. S. (editor), Introduction to Computer Architecture, Science Research Assoc., Chicago, 1975.

20. Wichmann, B. A., Algol 60 Compilation and Assessment, Academic Press, New York, 1973.

Appendix A: Test Program Specifications

A. I/O INTERRUPT KERNEL, FOUR PRIORITY LEVELS

A.1 INPUTS

An asynchronous I/O interrupt with associated status registers.

A.2 PROCESSING

The interrupt kernel will be activated by an I/O interrupt with priority level 0, 1, 2, or 3 from one of four devices. Actual interrupt processing will be simulated by counting the occurrences of each type of interrupt. Higher level interrupts will be able to preempt processing of lower priority interrupts. The interrupt handler must provide for resumption of processing of the preempted lower level interrupt from the point of preemption. As much processing as possible will be done with higher priority I/O interrupts enabled.

Figure 1 represents the functions to be performed. It is recognized that various architectures may provide automatic (hardware or firmware sypport for) features to carry out some of these functions. Appropriate use of such features is allowed. For example, when an interrupt occurs on the PDP-11, diasable can be achieved automatically be setting up ahead of time the appropriate processor priority level in the PSW of the device interrupt vector.

A.3 OUTPUTS

An updated count of interrupts by device.

A.4 CONSTRAINTS

This program need not be either position independent or reentrant.

The source files are available as specs.pub(c410gm20) at CMU-A.
If not send mail to George Mathew at CMU-A
December 10, 1976 DRAFT

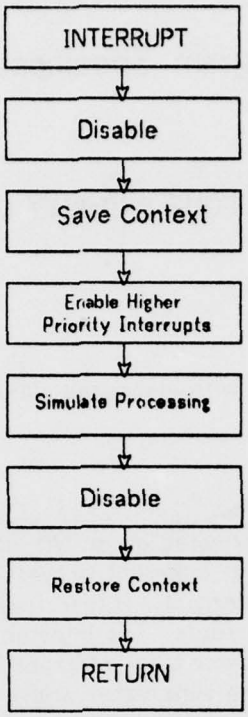


Figure 1: Priority I/O Kernel

B. I/O INTERRUPT KERNEL, FIFO PROCESSING

B.1 INPUTS

An asynchronous I/O interrupt with associated status registers.

B.2 PROCESSING

The interrupt kernel will be activated by an I/O interrupt from one of four devices which will be placed in a service queue for first-in-first-out (FIFO) processing. Actual interrupt processing will be simulated by counting the occurrences of each type of interrupt. Space should be provided to handle at least ten queued interrupts at one time.

Processing of queued interrupts shall be done with I/O interrupts enabled so that

interrupts will be taken and queued appropriately while previous interrupts are being processed. Before returning to the originally interrupted application program, a check shall be made to see if any interrupts remain queued, and these will be processed in FIFO order.

Figure 2a represents the functions to be performed. Appropriate use of an architectural feature to automatically provide some such function is allowed (see PDP-11 example in I/O interrupt kernel #1).

B.3 OUTPUTS

An updated count of interrupts by device.

B.4 CONSTRAINTS

This program need not be either position independent nor reentrant.

B.5 DISCUSSION

Consider the main processing loop, from "select next request (FIFO)" to the "Is queue empty?" test and back via the "NO" branch. During the processing from "Set run flag off" to "remove request from queue" through the NO branch of the text and back to "select next request" and "set run flag off". During this entire sequence interrupts are disabled; it is impossible for the "Is run flag on?" test to be executed at this point. The manipulation of the run flag can thus be safely moved out of the loop, producing figure 2b. This change should principally affect the M and R measures, although it could potentially lead to a smaller program. This change was not made, because it was felt that the measures for all three machines would be affected equally.

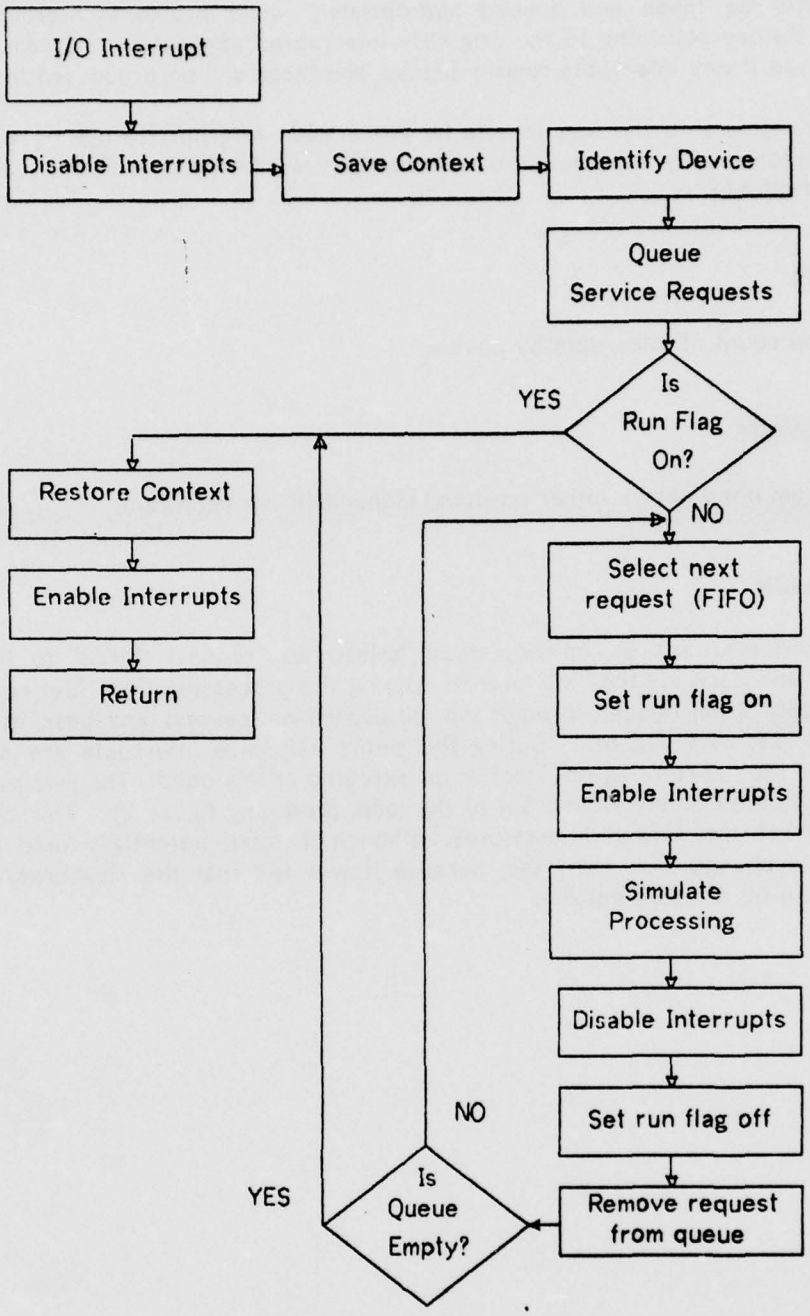


Figure 2a: Original FIFO Kernel

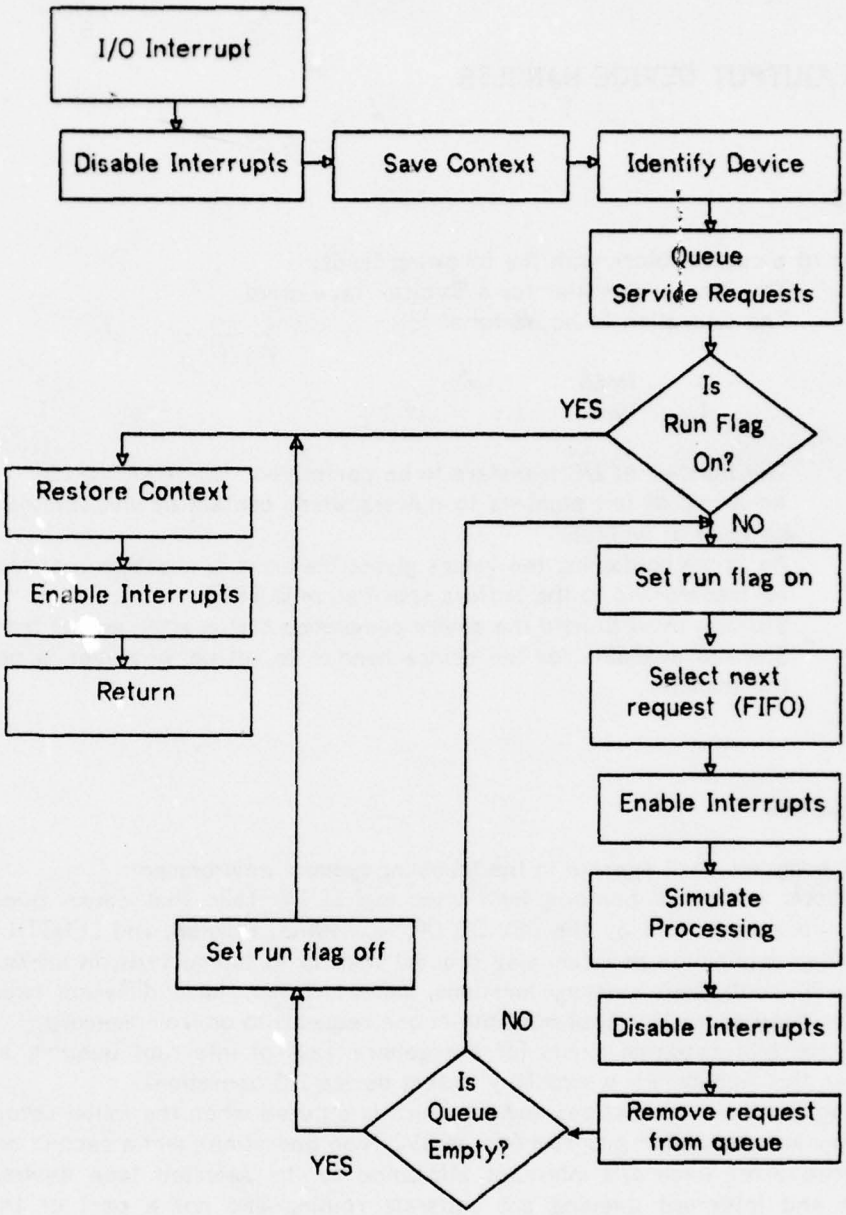


Figure 2b: Modified FIFO Kernel

C. INPUT/OUTPUT DEVICE HANDLER

C.1 INPUTS

A pointer to a control block with the following fields:

DEVICE — The device identifier for a "typical" tape drive.

O — The operation to be performed:

0 Read
1 Write

NO-TRANS — The number of I/O transfers to be performed. NO-TRANS ≤ 10

BUFFER — An array of ten pointers to buffers which contain or will contain the data to be read or written.

LENGTH — An array containing ten values giving the sizes in characters of the records to be transferred to the buffers specified in BUFFER.

STATUS — Storage used to hold the device completion status after an I/O transfer.

WORK — Storage available for the device handler to set up whatever is necessary for the transfer.

C.2 PROCESSING

This test program shall operate in the following general environment:

a. Applications programs perform high level logical I/O calls that cause queuing of the control block described by the DEVICE, OP, NO-TRANS, BUFFER, and LENGTH parameters. The calling application program may request that up to ten records, in arbitrary and not necessarily contiguous memory locations, and which may have different record lengths, be either read or written (but not both in one request) to or from memory.
b. A separate test program exists for the general task of interrupt queuing and handling (whereas this benchmark is explicitly to test device I/O operation).
c. This program has two sections, one of which is entered when the initial setup request is issued by an application program (via an SVC type operation), and a second section which is entered after each I/O interrupt attributed to the selected tape device. The SVC handler and interrupt queuing are separate routing and not a part of this program. Assume that they are taken care of.
d. The I/O devices handled by this program are all single density nine-track tape transports (pick any typical model tape transport for your machine).

After an I/O request is issued by an application program, and after the executive queues

the input control block, this test program is initiated and and it performs the following actions:

a. Check the status of the tape drive. If the device or its channel is busy, exit. If the device is not operable, branch to a dummy error routine. If the device is available, set up and initiate the requested transfer. For example, in the /370, fill in the fields of a CCW (or CCWs), set the CAW, and issue a SIO to the channel; in the PDP-11, fill in the device registers for the particular device and set the GO bit. Up to ten records may be written or read as a result of an I/O request.

b. After completion of the transfer, and a consequent interrupt, the device handler is reentered (via an interrupt vector, first level interrupt handler, etc.). The following processing is performed:
   i. The status information is stored into the STATUS entry of the control block. For example, in the /370, check the CSW, issue a read-status SIO, and store appropriate information in STATUS; in the PDP-11, identify the device, read its status registers, and store the information in STATUS.
   ii. If the device status indicates an unsuccessful transfer, abort any further processing and exit.
   iii. If the device status indicates a successful transfer, and if all requested transfers have been accomplished, then exit. Otherwise initiate the next requested transfer and then exit.

C.3 OUTPUTS

The completion status of the I/O transfer is returned via the STATUS entry of the control block.

C.4 CONSTRAINTS

The device handler is reentrant and self-relocating.

D. FAST FOURIER TRANSFORM

D.1 INPUTS

N	The number of data points. This is required to be an integral power of two in the range $0 \le N \le 2^{**}16$.
X	A vector holding the N samples as complex numbers. Each complex number has a real part and an imaginary part, both of which are 32-bit floating point numbers.
W	A vector holding the first N/2 powers of EXP(-2πi/N), where i**2 = -1. That is, W(j) = EXP(-2πi/N)**j.

WORK Pointer to auxiliary working storage.

D.2 PROCESSING

```
procedure FFT(N, X, W)
    GROUPS ← N
    do for PASS ← 0 by steps of 1 until log2(N)-1
        do for all ELEMENT such that 0 ≤ element ≤ N/2
            "generate complex addend"

            WEXP ← 0
            if PASS > 0
                then WEXP ← (((ELEMENT * N)/2)/ 2**PASS) MOD (N/2)
            end-if

            if WEXP ≠ 0
                then TEMP1 ← X(ELEMENT + N/2) * W(WEXP)
                else TEMP1 ← X(ELEMENT + N/2)
            end-if

            "generate 2 element entries in data vector"

            X1(ELEMENT) ← X(ELEMENT) + TEMP1
            X1(ELEMENT + N/2) ← X(ELEMENT) - TEMP1
        end-do
        if PASS < (log2(N) - 1)
            then
                "execute perfect card shuffle on data vector"

                P ← 2**PASS
                GROUPS ← GROUPS/2
                do for all I such that 0 ≤ I < GROUPS
                    do for all J such that 0 ≤ J < P
                        INDEX1 ← 2*P*I + J
                        INDEX2 ← P*I + J
                        X(INDEX1) ← X1(INDEX2)
                        X(INDEX1+P) ← X1(INDEX2 + N/2)
                    end-do
                end-do
            else
                do for all I such that 0 ≤ I < N
                    X← X1[I]
                end-do
        end-if
    end-do
```

D.3 PROGRAMMING CONSIDERATIONS

X1 is an auxiliary array of size N used to accomodate the card shuffle. Remember that both X and W are arrays of complex numbers.

D.4 OUTPUTS

A vector of FFT coefficients in X.

D.5 DISCUSSION

The original version of this specification included background on the history of the FFT, examples of the execution for small values of N, a short analysis of the algorithm, and some notes on performing complex arithmetic .

E. CHARACTER SEARCH

E.1 INPUTS

SRCHSTR	Pointer to a string of characters to be searched.
SRCHLNGTH	The length of that string. ≤ 2**15
SRCHARG	A pointer to a string of characters.
ARGLNGTH	The length of that string. ≤ 2**10
LOC	An integer return code
WORK	A ponter to any needed working storage

E.2 PROCESSING

SRCHSTR shall be searched to see if it contains a substring which exactly matches SRCHARG. If the search is successful the relative character position of the first occurrence of the substring shall be returned. If no match is found a negative value shall be returned.

E.3 OUTPUTS

If the search is successful, LOC is set to the relative character postion of the first occurrence of SRCHARG in SRCHSTR. Otherwise a negative value is stored in LOC.

E.4 CONSTRAINTS

The routine shall be reentrant and position independent.

```
procedure CHARSRCH(SRCHSTR, SRCHLNGTH, SRCHARG, ARGLNGTH, LOC, WORK)
    integer I
•
    LOC ← -1
    do for all I such that 0≤I≤SRCHLNGTH-SRCHARG or until LOC≠-1
        if the substring of SRCHSTR from I to I+ARGLNGTH-1 = SRCHARG
            then LOC ← I
        end-if
    end-do
```

F. BIT TEST, SET, OR RESET

F.1 INPUTS

F	Function code 1 test bit 2 set bit 3 reset bit
N	Relative bit number to be tested, 0 ≤ N ≤ 1000
A1	Pointer to a tightly packed bit string on an even word boundary
RC	Return code which is set to indicate the original status of the bit (0 or 1).
WORK	Pointer to any needed working storage.

F.2 PROCESSING

Bit number (N mod (word length)) of word (A1 + N/(word length)) is tested. If it is zero, then RC is set to zero; otherwise RC is set to one. If F is 2, the bit is then set to 1. If F is 3, the bit is then set to 0. For all othe values of F, the bit is unchanged. The bit string is assumed to begin at the first bit of location A1.

F.3 CONSTRAINTS

The routine shall be reentrant and position independent.

```
procedure BITTEST(F, N, A1, RC, WORK)
    integer ABIT, D

    ABIT ← A1 + N/(word length)
    D ← N mod (word length)

    if Dth bit at address ABIT = 1
        then RC ← 1
        else RC ← 0
    end-if

    if F = 2
        then Dth bit at address ABIT ← 1
        else if F = 3
                then Dth bit at address ABIT ← 0
             end-if
    end-if
```

G. RUNGE-KUTTA INTEGRATION

G.1 INPUTS

T0	Initial value of T, single precision floating point
Y0	Initial value of Y, single precision floating point
H	Interval of integration, single precision floating point
TMAX	Final value of T, single precision floating point
YMAX	final value of Y returned, single precision floating point
WORK	pointer to any needed working storage

G.2 PROCESSING

Given the differential equation

$$F(t,y) = t+y = dy/dt$$

and the initial conditions T0 and Y0, use a third order Runge-Kutta integration from T = T0 to T=TMAX to determine YMAX, using an integration interval H. All calculations are single precision floating point.

G.3 OUTPUTS

The value of Y at time TMAX is returned in YMAX.

G.4 CONSTRAINTS

The routine is position independent and reentrant.

```
procedure RUNGEKUTTA(T0, Y0, H, TMAX, YMAX, WORK)
    real K1, K2, K3

    YMAX ← Y0

    do for all T from T0 incremented in steps of H until T > TMAX
        K1 ← H * (T + YMAX)
        K2 ← H * (T + H/2 + Y + K1/2)
        K3 ← H * (T + 3*H/4 + Y + 3*K2/4)
        YMAX ← YMAX + 2*K1/9 + K2/3 + 4*K3/9
    end-do
```

H. LINKED LIST INSERTION

H.1 INPUTS

LISTCB	Pointer to a list control block, containing the entries	
	HEAD	Pointer to first node
	TAIL	Pointer to last node
	NUMENTRIES	Number of entries in list
NEWENTRY	Pointer to a new entry to be inserted.	

H.2 PROCESSING

List entries have the form

KEY	
NEXT	pointer to next entry
PREV	pointer to previous entry

All fields are one "word" (16 or 32 bits, depending on the machine) long. The KEY is a

signed 32-bit integer. The first node in the list is marked by a PREVPTR of zero, and the last is marked by a NEXTPTR of zero. NEWENTRY is inserted in order in the list. If there are duplicate keys the NEWENTRY is inserted after all matching entries. The list may be empty when the routine is called; this is indicated by a zero value in NUMENTRIES.

H.3 OUTPUTS

The new entry is updated to point forward and backwards. Both previous and next entries are updated to point to the new entry. NUMENTRIES is updated. If the new entry is a head or a tail then the pointers in the list control block are updated accordingly.

H.4 CONSTRAINTS

The routine is reentrant and position-independent.

```
procedure LISTINSERT(LISTCB, NEWENTRY)
    "the notation POINTER.FIELD is used to access a
        particular field of the structure pointed to by POINTER"

    pointer PRESENT

    if LISTCB.NUMENTRIES = 0
        then
            "list is empty, so initialize"

            LISTCB.HEAD ← LISTCB.TAIL ← NEWENTRY
            LISTCB.NUMENTRIES ← 1
            NEWENTRY.NEXT ← NEWENTRY.PREV ← 0

        else
            "list not empty"

            PRESENT ← LISTCB.HEAD
            LISTCB.NUMENTRIES ← LISTCB.NUMENTRIES + 1

            "determine position of new entry"

            while NEW.KEY ≥ PRESENT.KEY and PRESENT.NEXT ≠ 0 do
                PRESENT ← PRESENT.NEXT

            if PRESENT.PREV = 0 and NEW.KEY < PRESENT.KEY
                then
```

```
                    "new list head"

                    LISTCB.HEAD ← NEW
                    NEW.PREV ← 0
                    PRESENT.PREV ← NEW
                    NEW.NEXT ← PRESENT
                else
                    if NEW.KEY ≥ PRESENT.KEY
                        then
                            "new list tail"

                            PRESENT.NEXT ← LISTCB.TAIL ← NEW
                            NEW.NEXT ← 0
                            NEW.PREV ← PRESENT
                        else
                            "insert in middle"

                            NEW.NEXT ← PRESENT
                            NEW.PREV ← PRESENT.PREV
                            PRESENT.PREV ← NEW

                            "back up and link with predecessor"

                            PRESENT ← NEW.PREV
                            PRESENT.NEXT ← NEW
                    end-if
            end-if
    end-if
```

I. QUICKSORT

I.1 INPUTS

N	The number of records to be sorted. N ≤ 10000
REC	Pointer to the first element of an array of N+2 16-character records: R0, R1, ..., RN, RN+1. R0 is a dummy record with a low-valued key, and RN+1 is a dummy record with a high-valued key. Each record has a 7-character key found in positions 3 through 9, counting from 0.
M	integer parameter specifying the changeover point between QUICKSORT and a simple insertion sort. 5 ≤ M ≤ 20
WORK	pointer to any needed working storage.

I.2 PROCESSING

Records R0 to RN+1 are sorted into character collating sequence.

I.3 CONSTRAINTS

The routine is reentrant and position-independent.

```
procedure QUICKSORT(N, REC, M, WORK)
    integer L, R, I, J, K
    integr array STACK[0:2*f(N)-1]
    character string V

    REC[N+1] ← ∞
    L ← 1;    R ← N
    do forever
        I ← L;    J ← R+1;    V ← REC[L]
        do forever
            do I←I+1 until REC[I] ≥ V end-do
            do J←J-1 until REC[J] ≤ V end-do
            if J > I
                then swap REC[I] with REC[J]
                else go to end-first
            end-if
        end-do
end-first:
        swap REC[L] with REC[J]
        if both subfile sizes (J-L and R-J) ≤ M
            then
                if stack is empty
                    then go to end-outer
                    else pop L and R from stack
                end-if
            else
                if smaller subfile size ≤ M
                    then set L and R to lower and upper limits
                                of larger subfile
                    else
                        push lower and upper limits of larger
                                subfile onto stack
                        set L and R to limits of smaller subfile
                end-if
        end-if
    end-do
```

```
end-outer:
    do for I from N-1 to 1 in steps of 1
        if REC[I] > REC[I+1] then
            V ← REC[I];    J ← I+1
            do forever
                REC[J-1] ← REC[J];   J ← J+1
                if REC[J] ≥ V then go to end-last end-if
            end-do
end-last:   A[J-1] ← V
        end-if
    end-do
```

I.4 NOTES

F(N) is the maximum stack depth, and turns out to always be less than the natural logarithm of (N+1)/(M+2).

J. ASCII TO FLOATING POINT CONVERSION

J.1 INPUTS

N	Number of characters in the string
A1	Address of the character string; it may be assumed to be aligned on a word boundary.
A2	Address of a floating point number where the result will be placed. It may be assumed to be aligned on whatever boundary is appropriate for floating point numbers.

J.2 PROCESSING

Input is of the form

<optional sign> <decimal digits> . <decimal digits>

where either set of decimal digits may be omitted. You may assume that the input is correctly formatted and that there are no extra characters beyond the end of the number.

J.3 CONSTRAINTS

This routine is reentrant and position-independent.

```
procdure AFP(N, A1, A2)
    integer NUMBER, POSITION
    real RESULT, DIVISOR
    boolean ISNEGATIVE

    ISNEGATIVE ← false
    POSITION ← 0
    if first character of A1 is a sign character
        then
            if sign character is "-"
                then ISNEGATIVE ← true
            end-if
            POSITION ← 1
    end-if
    NUMBER ← integer equivalent of characters POSITION to J-1 of A1
                where character J of A1 is "."
    RESULT ← floating point equivalent of NUMBER

    "the following two steps can be done in parallel, if desired"

    NUMBER ← integer equivalent of characters J+1 to N of A1
    DIVISOR ← floating equivalent of 10**(N-J)

    A2 ← RESULT + (floating equivalent of NUMBER) / DIVISOR
```

K. BOOLEAN MATRIX TRANSPOSE

K.1 INPUTS

A1	Pointer to a word of storage
A2	bit number within word A1 where the matrix begins
N	size of the boolean matrix

K.2 PROCESSING

Transpose a tightly-packed bit matrix. All the bits of the first row are contiguous, and are followed immediately by the bits of the second row.

```
procedure BMT(N, A1, A2)
    integer I, J
    boolean B[1:N,1:N] beginning at bit A2 of word A1

    do for all I and J such that (1 ≤ J ≤ N) and (J+1 ≤ I ≤ N)
        swap B[I,J] and B[J, I]
    end-do
```

L. VIRTUAL MEMORY SPACE EXCHANGE

L.1 PROCESSING

Write a miniature supervisor call handler which provides the two functions "call" and "return". CALL is function 0, RETURN is function 1 (i.e. SVC 0, SVC 1 on the /370, TRAP 0, TRAP 1 on the PDP-11). In the following, "segment" means whatever IBM means by a segment on the /370, whatever INTERDATA means by the term on the 8/32, and one of 8 eight-kilobyte pages on the PDP-11. Parameters will be passed according to whatever calling convention is used for ordinary subroutines on the machine in question.

CALL takes two parameters. CALLEE is an integer in the range 0 ≤ CALLEE ≤ 255. It is the index of an entry in a table of address spaces maintained within the supervisor. (An "address space" is a set of segments). PARAMETER is an address in user space. CALL saves enough information to restore the entire state of the caller (e.g., all the fixed point and floating point registers, the program counter, and so on). It then establishes addressibility for the address space indicated by CALLEE. One of the segment pointers in the address space will be marked in some way to indicate that it is null. It is replaced by a pointer to the segment containing the word addressed by PARAMETER. It then begins executing the address space at some arbitrary point.

RETURN takes no parameters. It restores the environment active before the previous call.

Calls may be nested up to eight deep; you need not check for this bound being exceeded. You need not check for a RETURN with no matching CALL. You need not build the address space table, but merely describe its format. You need not let an address space call itself recursively, nor need you check for this condition.

L.2 CONSTRAINTS

This program need not be either position-independent nor reentrant.

Appendix B : CFA Test Data Specifications

This document specifies the test data which the routines must satisfactorily process to certify them as debugged. Mail the final listings of your test programs along with printout demonstrating their correct operation to Sam Fuller. The data sets marked * will be used to compute the S, M and R measures on the ARF simulator. Those data sets marked + will be used to compute these measures by hand in the event that the simulator is not available in time. Where the program specifications provide pointers, this document gives the values pointed at by the pointer, where appropriate. This is only for convenience and does NOT mean the parameters should all be passed as these values rather than as pointers.

A. I/O kernel, Four Priority Levels.

Test scenario: An interrupt is received on the lowest priority level. While this interrupt is being processed a second interrupt is received on the highest level. Processing of the lower priority interrupt is intervened by the higher priority interrupt. Processing then continues until both interrupts have been processed, and the machine restored to its pre-interrupt state.

B. I/O Kernel, FIFO Processing.

An interrupt is received on one of the four devices. While this interrupt is being processed, another is received on another device. The second interrupt is queued, and processing continues until both interrupts are processed, and the machine restored to its pre-interrupt state.

C. I/O Device Handler.
The handler is called with the following inputs:

Device:	tape
Op:	0 (read)
No-trans:	2
Buffer:	addr1, addr2, 0, ...
Length:	120, 120, 0, ...
Status:	0
Working-Storage:	pointer

The handler then sets up the two reads, and performs the necessary processing to perform the reads and then returns the completion status for the reads.

D. Large FFT. (*See attached appendix for illustrations of how the FFT procedure should transform the data, i.e. the X vector).*

In the following tests, the W vector will be as follows:

W(0) = (1.0, 0.0)
W(1) = (0.0245412, 0.9996988) = (cos π/128, sin π/128)
W(2) = $W(1)^2$
...
...
W(128) = $W(1)^{128}$ = (-1.0, 0.0)

*1) Test case: *Step Function.* Parameters are:

i. X:
X(0) = (1.0, 0.0)
X(1) = (1.0, 0.0)
.....
X(127) = (1.0, 0.0)
X(128) = (0.0, 0.0)
.....
.....
X(255) = (0.0, 0.0)
ii. W: *W vector above*
iii. N: 256
iv. WORK

2) Test case: *i sin πs.* Parameters are:

i. X:
X(0) = (0.0, 0.0)
X(1) = (0.0, 0.0)
.....
X(63) = (0.0, 0.0)
X(64) = (-1.0, 0.0)
X(65) = (0.0, 0.0)
.....
X(127) = (0.0, 0.0)
X(128) = (1.0, 0.0)
X(129) = (0.0, 0.0)
.....
X(191) = (0.0, 0.0)
X(192) = (-1.0, 0.0)
X(193) = (0.0, 0.0)
....
X(255) = (0.0, 0.0)
ii. W: *W vector above*
iii. N: 256
iv. WORK

+3) Test case: *Small Step Function.* Parameters are:

i. X: X(0) = (1.0, 0.0)
X(1) = (1.0, 0.0)
X(2) = (0.0, 0.0)
X(3) = (0.0, 0.0)
X(4) = (1.0, 0.0)
.....
ii. W: W(0) = (1.0, 0.0)
W(1) = (0.3827, 0.9239) = (cos π/8, sin π/8)
W(2) = $W(1)^2$
.....
W(4) = $W(1)^4$ = (0.0, 1.0)
.....
W(8) = $W(1)^8$ = (-1.0, 0.0)
iii. N: 16
iv. WORK

E. Character Search.

1) Test case: *Find match.* Parameters are:

i. SRCH-STR: "Monday, June 7th, 1976"
ii. SRCH-LNGTH: 22
iii. SRCH-ARG: "day"
iv. ARG-LNGTH: 3
v. LOC: *Expected return value* = 3
vi. WORK

**2) Test case: *No match.* Parameters are:

i. SRCH-STR: "Carnegie-Mellon U"
ii. SRCH-LNGTH: 17
iii. SRCH-ARG: "CMU"
iv. ARG-LNGTH: 3
v. LOC: *Expected return value* = -1
vi. WORK

*3) Test case: *Match at attempt n (n > 1).* Parameters are:

i. SRCH-STR: "Day in, Day out"
ii. SRCH-LNGTH: 15
iii. SRCH-ARG: "Day out"
iv. ARG-LNGTH: 7
v. LOC: *Expected return value* = 8
vi. WORK

4) Test case: *Match at beginning of string.* Parameters are:

i. SRCH-STR: "abcd"
ii. SRCH-LNGTH: 4

iii. SRCH-ARG: "ab"
iv. ARG-LNGTH: 2
v. LOC: *Expected return value* = 0
vi. WORK

5) Test case: *Fail match at end of string*. Parameters are:

i. SRCH-STR: "efgh"
ii. SRCH-LNGTH: 4
iii. SRCH-ARG: "hx"
iv. ARG-LNGTH: 2
v. LOC: *Expected return value* = -1
vi. WORK

6) Test case: *Match first occurence*. Parameters are:

i. SRCH-STR: "Day in, Day out"
ii. SRCH-LNGTH: 15
iii. SRCH-ARG: "Day"
iv. ARG-LNGTH: 3
v. LOC: *Expected return value* = 0
vi. WORK

7) Test case: *Deal with 0 length search string*. Parameters are:

i. SRCH-STR: "Day in, Day out"
ii. SRCH-LNGTH: 0
iii. SRCH-ARG: "Day"
iv. ARG-LNGTH: 3
v. LOC: *Expected return value* = -1
vi. WORK

8) Test case: *Deal with 0 argument search string*. Parameters are:

i. SRCH-STR: "Day in, Day out"
ii. SRCH-LNGTH: 15
iii. SRCH-ARG: "Day"
iv. ARG-LNGTH: 0
v. LOC: *Expected return value* = 0
vi. WORK

*9) Test case: *Match at attempt n (n > 1)*. Parameters are:

i. SRCH-STR: "Saving hot water saves gas. So take shorter showers,
or run a little less hot water in the tub.
Do full loads in your dishwasher and washing machine.
Fix leaky faucets. It helps to keep your gas water
heater at the normal setting or lower."

Note: Each line break in the above string is a space.

ii. SRCH-LNGTH: 240
iii. SRCH-ARG: "water heater"
iv. ARG-LNGTH: 12
v. LOC: *Expected return value* = 196
vi. WORK

F. Bit Test, Set, or Reset.

All tests given here count from the left hand end of the string. However, the results expected below (i.e. returned code) will be the same for bits counted in either direction. The programmer must verify the results of a routine which counts bits from the low order end of the word. The following bit string is used in all of the next six tests: Consider the bits to be grouped in 16 bit *words*:

1010011100101111 0111001010011100 1110111000000110 1100101110111000

1) Test case: *Test 1 bit*. Parameters are:

i. F: 1
ii. N: 19
iii. A1: *Bit string above*
iv. RC: *Expected return value*: 1
v. WORK

2) Test case: *Test 0 bit*. Parameters are:

i. F: 1
ii. N: 35
iii. A1: *Bit string above*
iv. RC: *Expected return value*: 0
v. WORK

**3) Test case: *Set 0 bit*. Parameters are:

i. F: 2
ii. N: 21
iii. A1: *Bit string above*
iv. RC: *Expected return value*: 0
Also expect: Word 1 = 0111011010011100
v. WORK

4) Test case: *Set 1 bit*. Parameters are:

i. F: 2
ii. N: 34
iii. A1: *Bit string above*
iv. RC: *Expected return value*: 1
v. WORK

5) Test case: *Clear 1 bit*. Parameters are:

i. F: 3
ii. N: 0
iii. A1: *Bit string above*
iv. RC: *Expected return value*: 1
Also expect: Word 0 = 0010011100101110
v. WORK

**6) Test case: *Clear 0 bit*. Parameters are:

i. F: 3
ii. N: 16
iii. A1: *Bit string above*
iv. RC: *Expected return value*: 0
v. WORK

G. Runge-Kutta Integration.

**1) Test case: *Accuracy*. Parameters are:

i. t_0: 0.0
ii. y_0: 0.0
iii. h: 0.0078125
iv. t_{max}: 10.0
v. y: *Expected return value*: 22188
vi. WORK

2) Test case: $t0 = t_{max}$. Parameters are:

i. t_0: 1.0
ii. y_0: 0.0
iii. h: 0.01
iv. t_{max}: 1.0
v. y: -10.0 (*Initial value*) *Expected return value*: 0.010100333...
vi. WORK

**3) Test case: *Straight line*. Parameters are:

i. t_0: 0.0
ii. y_0: -1
iii. h: 0.5
iv. t_{max}: 1.5
v. y: *Expected return value*: -3
vi. WORK

H. Linked List Insertion.

1) Test case: *Insert first key in empty list.* Parameters are:

i. LIST-CB: *Contains*: (0, 0, 0)
ii. NEW-ENTRY: (111, 0, 0)

2) Test case: *Insert at head of list.* Parameters are:

i. LIST-CB: *Contains*: (?, ?, 1)
ii. NEW-ENTRY: (45, 0, 0)

3) Test case: *Append to list.* Parameters are:

i. LIST-CB: *Contains*: (?, ?, 2)
ii. NEW-ENTRY: (150, 0, 0)

4) Test case: *Insert in middle of list.* Parameters are:

i. LIST-CB: *Contains*: (?, ?, 3)
ii. NEW-ENTRY: (100, 0, 0)

5) Test case: *Duplicate key at head of list.* Parameters are:

i. LIST-CB: *Contains*: (?, ?, 4)
ii. NEW-ENTRY: (45, 0, 0)

6) Test case: *Duplicate key at tail of list.* Parameters are:

i. LIST-CB: *Contains*: (?, ?, 5)
ii. NEW-ENTRY: (150, 0, 0)

7) Test case: *Duplicate key in middle of list.* Parameters are:

i. LIST-CB: *Contains*: (?, ?, 6)
ii. NEW-ENTRY: (111, 0, 0)

**8) Test case: *Insert in middle of list.* Parameters are:

i. LIST-CB: *Contains*: (?, ?, 7)
ii. NEW-ENTRY: (75, 0, 0)

Note: As a second sequence of 8 tests use the keys in reverse order: 75, 111, 150, 45, 100, 150, 45, 111.

I. Quick Sort.

*1) Test case: *General sort.* Parameters are:

i. N: 30
ii. REC: *See table below*
iii. M: 6
iv. WORK

Note: When sorted, the numbers in positions 0 - 3 of each record will be in ascending order of magnitude.

```
 "0       **********",
"48 mcwso++++++++","24 frktb++++++++","68 savwu++++++++",
"36 jjklv**********","88 vbvbs**********","120zoupm**********",
"40 jsgcz++++++++","4  aercr++++++++","100xjvgc++++++++",
"104ymljz**********","64 rooxc**********","76 tasgo**********",
"116zkbuk++++++++","92 vcobr++++++++","32 jhkpx++++++++",
"56 qmfzy**********","80 tzwkt**********","44 ldvuj**********",
"60 qocwf++++++++","84 uwbyb++++++++","20 cydhp++++++++",
"28 iwyun**********","72 sohrg**********","12 bwhic**********",
"52 qgggj++++++++","16 csqcw++++++++","96 vrkzw++++++++",
"112zhrhb**********","108yqmba**********","8  alvoz**********",
 "124zzzzz++++++++",
```

+2) Test case: *General sort.* Parameters are:

i. N: 6
ii. REC: *See table below*
iii. M: 2
iv. WORK

```
 "0       ++++++++",
"48 mcwso**********","24 frktb**********","68 savwu**********",
"36 jjklv++++++++","88 vbvbs++++++++","120zoupm++++++++",
 "124zzzzz**********",
```

J. ASCII to Floating Point Conversion Routine.

*+1) Test case: *Ordinary (representable) F.P. number.* Parameters are:

i. N: 7
ii. A1: "10.0625"
iii. A2: *Expect:* (#041041 #000000 on PDP 11)
Expect: (? on IBM 370, Interdata 8/32)

2) Test case: *No <integer> part.* Parameters are:

i. N: 5
ii. A1: ".0625"
iii. A2: *Expect:* (#037200 #000000 on PDP 11)
Expect: (? on IBM 370, Interdata 8/32)

3) Test case: *No <fraction> part.* Parameters are:

i. N: 3
ii. A1: "10."
iii. A2: *Expect*: (#041040 #000000 on PDP 11)
Expect: (? on IBM 370, Interdata 8/32)

4) Test case: *Ordinary (non-representable) F.P. number.* Parameters are:

i. N: 4
ii. A1: "0.01"
iii. A2: *Expect*: (#036413 #153412 on PDP 11)
Expect: (? on IBM 370, Interdata 8/32)

5) Test case: *Ordinary F.P. number.* Parameters are:

i. N: 9
ii. A1: "0.5009766" (= $2^{-10} + 0.5$)
iii. A2: *Expect*: (#040000 #040000 on PDP 11)
Expect: (? on IBM 370, Interdata 8/32)

6) Test case: *Optional sign* <->. Parameters are:

i. N: 4
ii. A1: "-2.0"
iii. A2: *Expect*: (#140400 #000000 on PDP 11)
Expect: (? on IBM 370, Interdata 8/32)

7) Test case: *Optional sign* <+>. Parameters are:

i. N: 4
ii. A1: "+2.0"
iii. A2: *Expect*: (#040400 #000000 on PDP 11)
Expect: (? on IBM 370, Interdata 8/32)

8) Test case: *Floating point 0.* Parameters are:

i. N: 3
ii. A1: "0.0"
iii. A2: *Expect*: (#000000 #000000 on PDP 11)
Expect: (? on IBM 370, Interdata 8/32)

K. Boolean Matrix Transpose.

+1) Test case: *Array starting off a word boundary.* Parameters are:

i. N: 4
ii. A1:

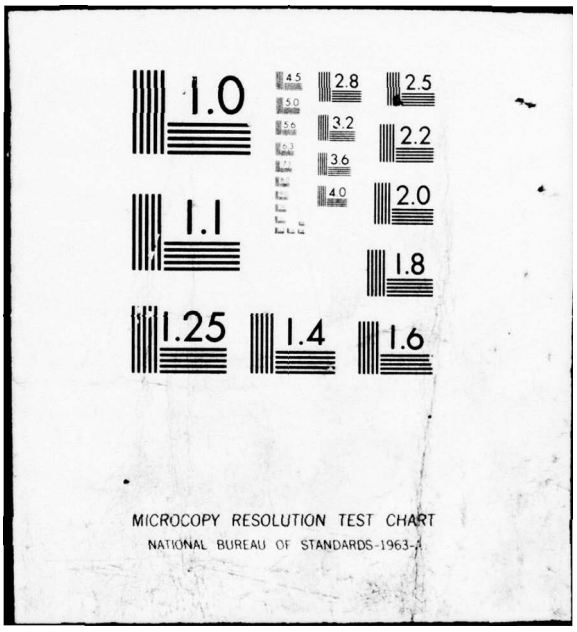
1 1 0 1 1 1 1 1

AD-A049 482 ARMY ELECTRONICS COMMAND FORT MONMOUTH N J F/G 9/2
COMPUTER FAMILY ARCHITECTURE SELECTION COMMITTEE FINAL REPORT. --ETC(U)
SEP 77 W E BURR, S H FULLER, P S SHAMAN

UNCLASSIFIED ECOM-4528 NL

2 OF 2
AD
A049 482

END
DATE
FILMED
3-78
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

```
1 0 1 1       1 0 0 0
1 0 0 0   =>  0 1 0 0
1 0 0 1       1 1 0 1
```

iii. A2: 4

*2) Test case: *Array > 1 word in length.* Parameters are:

i. N: 9

ii. A1:

```
0 0 1 0 1 0 0 1 1       0 1 1 0 1 0 1 0 1
1 0 0 1 0 1 1 1 0       0 0 1 1 1 0 0 0 0
1 1 1 0 0 0 0 1 0       1 0 1 1 0 1 1 0 0
0 1 1 1 0 0 1 0 1       0 1 0 1 1 0 1 0 1
1 1 0 1 1 1 0 0 0   =>  1 0 0 0 1 1 1 0 1
0 0 1 0 1 1 1 0 0       0 1 0 0 1 1 0 1 1
1 0 1 1 1 0 1 1 1       0 1 0 1 0 1 1 0 1
0 0 0 0 0 1 0 1 0       1 1 1 0 0 0 1 1 1
1 0 0 1 1 1 1 1 0       1 0 0 1 0 0 1 0 0
```

iii. A2: 0

3) Test case: *One dimensional array.* Parameters are:

i. N: 1

ii. A1:

```
1 0       1 0
1 0   =>  1 0
```

iii. A2: 4

L. Virtual Memory Space Exchange.

Will not be run on actual machines.

Appendix C: Calling Sequence Conventions

This appendix assumes knowledge of the instruction sets of the candidate architectures.

In order to debug the benchmark programs, it was necessary to write driver programs to call the benchmarks as subroutines. This made it necessary to establish conventions for the calling sequences that would be used to invoke the benchmarks.

At first it was decided to use the calling conventions of the language used to write the drivers. On the 8/32 and the /360 the drivers were written in FORTRAN; on the 11 it was written in BLISS-11. For the canonical program

```
RTN(A1,...An)
```

the calling sequence for the PDP-11 was

```
        MOV     address of downward-growing stack,R6
        ...
        MOV     A1,-(R6)        ; stack value of first argument
        ...
        MOV     An,-(R6)
        JSR     PC,RTN          ; push program counter
                                ; and jump to subroutine
```

On entry to a subroutine, register 6 pointed to a word containing the address of the instruction following the JSR. The address of the last argument was in the location at offset 2 from the register 6, the next-to-last at offset 4, and so on.

For the IBM 360 the calling sequence was

```
        L       13,=A(SAVE)     load address of save area in R13
        L       1,=A(ARGS)      load address of argument list in R1
        L       15,=A(RTN)      get address of routine to R15
        BALR    14,15           jump to subroutine
        ...
ARGS    DC      A(A1)   full word, address of argument 1
        ...
        DC      A(An)
```

Register 14 contained the address of the instruction following the subroutine call; register 15 contained the address of the first instruction of the subroutine and could be used as a base register. Register 1 contained a pointer to a list of addresses of arguments. The address of the first argument was in the word pointed to by register 1; the address of the second argument was in the word at offset 4 from the word pointed to by register 1, and so on.

Register 13 pointed at the base of an area used to store the general purpose registers. The actual convention for the use of the area pointed to by register 13 is fairly complicated, and involves chaining together pointers to such save areas. It was stated fairly early that we did not want to impose the full burden of this convention on the programmers. They could simply consider register 13 to be a pointer to a 16-word block where they could save the general registers[1].

For the Interdata 8/32, the calling sequence was

```
L       12,address of start of downward-growing stack
...
BAL     15,RTN
DC      X'<2*N+2>'
DAC     A1      address of first argument
...
DAC     An
next instruction
```

Register 12 points to the top of a downward-growing stack; the conventional use of this stack is to subtract some constant from the stack pointer and use positive offsets from the stack pointer as temporary storage[2]. Register 15 contains a pointer to the halfword immediately following the subroutine call. This halfword contains information relating to the number of parameters. The halfword is followed by a set of fullwords containing addresses of parameters. Because the instruction and the following halfword of information need only be aligned on a halfword boundary, while the address constants must be aligned on a fullword boundary, the following two situations might occur. (A vertical bar, |, denotes a fullword boundary, while a colon, :, denotes a halfword boundary).

```
-------------------------------------------------------------
|       : BAL  | DC   : empty | DAC A1       | DAC A2       |
-------------------------------------------------------------
|       :      | BAL  : DC    | DAC A1       | DAC A2       |
-------------------------------------------------------------
```

---

[1] In the standard calling convention, one stores the registers starting at offset 12 from register 13 via

```
STM     14,12,12(13)
```

[2] Unlike on the PDP-11, this is a software convention; on the PDP-11, the register 6 stack is used by the hardware.

In both cases, register 15 points to the DC constant. In the first case, the first address is at offset 2 from register 15. In the second, it is at offset 4. It is necessary for the program to force register 15 to point to a consistent place; the instruction

```
OHI     15,2    OR halfword immediate
```

as the first instruction of the subroutine forces register 15 to point to the halfword immediately preceding the "DAC A1". Thus the address of the first parameter is contained in the word at offset 2 from register 15; the second is at offset 6, and so on. It was later decided that the existing calling sequences were unfairly penalizing some of the machines whose drivers imposed complicated or inappropriate calling sequences. In particular,

a. On the PDP-11, it was usually unnecessary to use the "work area" parameter, as the register 6 stack provided a work area. On the 8/32, the software stack based on register 12 served a similar function. Clearly, on the /370 it would be possible to implement a software stack like on the 8/32; it seemed unfair to penalize the /370 on the basis of a software convention.

b. On the PDP-11, the BLISS driver was passing the values of parameters rather than their addresses. This eliminated a level of indirection which perhaps unfairly penalized the /370 and the 8/32, which lack indirection.

To relieve these problems, the calling sequences were modified. Programmers were permitted to use the area pointed to by register 13 as their work area on the /370. All "input" parameters, those examined by the routines but not overwritten, were passed by value rather than by reference. Those parameters specified as being addresses in the program specifications, as well as those parameters into which results were to be stored, were still passed by reference. For example, the CHARACTER SEARCH program, E, had the parameters

1) SRCHSTR, the address of a character string

2) SRCHLNG, the length of SRCHSTR

3) SRCHARG, the address of another string

4) ARGLNG, its length

5) LOC, a place to store the result

6) WORK, a pointer to extra working storage

Under the original conventions, the argument list might look like

```
ARGLST   DC      A(SRCHSTR)
         DC      A(SRCHLNG)
         DC      A(SRCHARG)
         DC      A(ARGLNG)
         DC      A(LOC)
         DC      A(WORK)
         ...
SRCHSTR  DC      C'Search string'
SRCHLNG  DC      F'13'
SRCHARG  DC      C'Arg'
ARGLNG   DC      F'3'
LOC      DS      F
WORK     DS      ...
```

Under the new conventions, this became

```
ARGLST   DC      A(SRCHSTR)
         DC      F'13'
         DC      A(SRCHARG)
         DC      F'3'
         DC      A(LOC)
         DC      A(WORK)
         ...
SRCHSTR  DC      C'Search string'
SRCHARG  DC      C'Arg'
LOC      DS      F
WORK     DS      ...
```

Appendix D: S, M, and R Calculation Sheets

. This appendix contains the actual S, M, and R measures for each instruction, addressing modes, and register format for the three final three candidate architectures. The instructions preceding the calculation sheets assumes the computations will be done manually but in fact the whole process was automated. If test programs were run on the ISP simulator (which the majority were), the output from the ISP simulator was read directly by the summary calculation program, minimizing the chance of making a clerical error. If the instruction counts had to be done by hand, then the instruction counts were typed into a disk file and then the file was input to the summary calculation program.

The calculation of the appropriate M measure for each instruction and each addressing mode is reasonably straightforward and fairly obvious. The R measure is another story, since the R measure for each case is determined by effectively microcoding that instruction and addressing mode in terms of the anonical microprocessor described in Section 3.2. Since this amounts to programming, two different microcoders may implement the same instruction in two different ways. To minimize this source of error, a single individual computed the R measures for all three final candidates (W. E. Burr), making every effort to be consistent across the three architectures, and the calculations were reviewed by the respective architecture chairmen.

The following calculations of the R measure for a basic instruction interpretation cycle and address calculation for each of the three architectures illustrate the approach used to determine the actual R measures given here in this appendix.

July 7, 1976
D.A. Lamb

Instructions for Completing the IBM S/370 Calculation Sheet

To assist you in the computation of the S, M, and R measures for the IBM 370, we have included here a set of **calculation sheets** (8 pages) and a liberal supply of **work sheets**. The work sheets are designed to be directly attached to the listing of your test program and have entries for each line of your program. Use as many work sheets as you have pages of code in your listing. The calculation sheets will be used to enumerate S, M, and R measures via addressing modes and instruction type.

The principle use of the work sheets will be to get accurate counts of instruction executions. The calculation sheets will be used to determine the final S, M, and R measures. Later we will indicate how to obtain these measures from your work sheets directly, but this will be used primarily as a check on your calculations.

1. You will need to refer to most of the calculation sheets at the same time, so you should work at a desk with enough space to spread out all of the sheets at once. You can lay aside the summary calculation sheet (page 1) since you will not need it until the end. You will also need the CFA Test Data Specifications of June 16, 1976.

2. Attach the work sheets to your program. The work sheets have a line for each line of your program, and a column for the instruction type (Mode), the number of times the program is executed (N), the storage measure (S), the memory transfer measure (M), the register transfer measure (R), and the products N*M and N*R.

3. Go through your program to determine how often each instruction will be executed, using the test data items marked with a plus (+) in the specifications. Fill in the N column of the work sheet with these numbers. Where there is more than one set of test data for a particular program, determine the number of times each instruction is executed for each set of data, add the results together, and write the sum in the N column. If your program is simple enough, you may be able to do an analysis to determine these numbers in terms of the size of the test data and some simple characteristics (such as the number of zeros in the matrix for the boolean matrix transpose program). Failing that, you may need to hand-simulate the operation of your program on the test data.

4. Perform steps 5 to 12 for each instruction in the program. Each step is labeled with the name of the sheet on which the number calculated in the step is to be written.

5. work sheet: Determine the type of the instruction (RR, RX, SI, RS, or SS) and mark the type in the Mode column of the work sheet.

6. calculation sheet: Find the row of the Instruction Format Table (page 2) corresponding to the type determined in step 5. Add one to the "number of

static occurrences" column of that row, and add the number in the N column of the work sheet to the "no. of times executed" column of the row.

7. work sheet: Copy the number from the "S/INST" column of the row in step 6 to the S column of the work sheet.

8. calculation sheet: For each effective address calculation in the instruction, find the appropriate row in the Effective Address Calculation table (page 2). RR instructions have no effective address calculations, RX, RS, and SI have one effective address calculation, and SS have two. The table has a separate row for pure displacement (no registers), base plus displacement (one register), and base, displacement, and index (two registers). Add one to the "no. of times executed" column of the row.

9. calculation sheet: Find the row of the Instruction Table (pages 3 to 7) corresponding to the instruction. Add the number in the N column of the work sheet to the "no. of times executed" column of this row. If the "R/INST" and "M/INST" columns of this row are blank, mark the row (preferably in a colour that stands out), write zeros in these two columns, and on a separate sheet make note of the fact that the instruction was missing.

10. work sheet: Add the number in the "R/INST" column of the row found in step 9, the number of the "R/CALC" column of the row found in step 8, and the number in the "R/INST" column of the row found in step 6. Write this sum in the R column of the work sheet.

11. work sheet: Add the number in the "M/INST" column of the row found in step 9 and the number in the "M/INST" column of the row found in step 6. Write this sum in the M column of the work sheet.

12a. calculation sheet: If the entry in the Instruction Table (found in step 9) has an asterisk (*), the M and R measures depend on some other count associated with the instruction. For example, a STM (store multiple) instruction costs 4 bytes in the M measure for every register stored. For each such starred instruction, look up its entry in the "Miscellaneous M and R table" (page 8), calculate the appropriate "what to count" number, multiply by the entry in the N column of the work sheet, and add this number to the "count" column of the table.

12b. work sheet: Multiply the "what to count number" determined in step 12a by the number in the "M/COUNT" entry used in that step, and add this product to the M column of the work sheet.

When you have completed these calculations for the entire program, compute the totals in the tables as outlined in steps 13 to 19.

13. calculation sheet: For each entry of the Instruction Table (page 3 through 7), multiply the number in the "no. of times executed" column by the number in the "M/INST" column and write the product in the "M subtotal" column. Multiply the number in the "no. of times executed" column by the number in the "R/INST" column and write the product in the "R subtotal" column.

14. calculation sheet: For each page of the Instruction table, calculate the subtotals of the "number of times executed", "M subtotal", and "R subtotal" columns, fill in the subtotals at the bottom of the page (C_{P1}, M_{P1}, R_{P1} for instruction page 1, and so on), and write these subtotals in the space provided on the Summary Calculation page (page 1).

15. calculation sheet: On the Instruction Format Table (page 2), add up the "S subtotals", "number of times executed", "M subtotal", and "R subtotal" columns to give S_I, C_I, M_I, and R_I. Fill in the corresponding entries on the Summary Calculation page.

16. calculation sheet: On the Effective address Calculation table, add up the columns giving C_{eac} and R_{eac}, and copy these to the summary page.

17. calculation sheet: Add up the entries on the summary page to give the S, M, and R measures.

18. calculation sheet: Perform the consistency checks given on the summary page. You should get C_I equal to $C_{P1} + \ldots + C_{P5}$, and C_{eac} equal to $C_{RX} + 2C_{SS}$.

19. work sheet: Multiply the N column of the work sheet by the M column and write the result in the NM column. Multiply the N column by the R column and write the result in the NR column. Calculate the sum of the NM column; this should be the same as the final M measure. Similarly, the sum of the NR column should be the same as the final R measure.

June 30, 1976

IBM S/370 CALCULATION SHEET

S, M, and R Measures

SUMMARY CALCULATIONS

Test Program: _____

Programmer: _____

Date: _____

S MEASURE

Instruction subtotal, S_I: _____

Constants and local variables: _____

Temporary workspace: _____

S [ ]

M MEASURE

Instruction format subtotal M_I: _____

Instruction page subtotal, M_{P1}: _____

M_{P2}: _____

M_{P3}: _____

M_{P4}: _____

M_{P5}: _____

Miscellaneous subtotal, M_M: _____

M [ ]

R MEASURE

Instruction format subtotal, R_I: _____

Instruction page subtotal, R_{P1}: _____

R_{P2}: _____

R_{P3}: _____

R_{P4}: _____

R_{P5}: _____

Miscellaneous subtotal, R_M: _____

Address calculation subtotal, R_{eac}: _____

R [ ]

CONSISTENCY CHECKS ON CALCULATIONS

You can use the following identities to provide some measure of assurance you have not made a clerical error in tabulating the S, M, and R measures:

(1) $C_{P1} + C_{P2} + C_{P3} + C_{P4} + C_{P5} = C_I$

(2) $C_{eac} = C_{RX} + 2C_{SS}$

INSTRUCTION FORMAT TABLE

INSTRUCTION FORMAT	NO. STATIC OCCURANCES	S/INST	S SUBTOTALS	NO. TIMES EXECUTED	M/INST	M SUBTOTAL	R/INST	R SUBTOTAL
RR		2		C_{RR}:	2		8	
RX, RS, SI, and S		4		C_{RX}:	4		16	
SS		6		C_{SS}:	6		12	
Instruction format subtotal			S_I:	C_I:		M_I:		R_I:

EFFECTIVE ADDRESS CALCULATION TABLE

	TYPE OF ADDRESS CALCULATION	NO. TIMES EXECUTED	R/CALC	R SUBTOTALS
displacement only:	$B_2 = \emptyset$, $X_2 = \emptyset$, and $B_1 = \emptyset$		5	
base + displacement:	Either $B_1 > \emptyset$, $B_2 > \emptyset$, or $X_2 > \emptyset$		8	
base + displacement + index:	$B_2 > \emptyset$ and $X_2 > \emptyset$		17	
Address calculation subtotal		C_{eac}:		R_{eac}:

IBM S/370 Calculation Sheet

Test Program:_____
Programmer:_____
Date:_____

IBM S/370 CALCULATION SHEET

Test Program: _____

Programmer: _____

Date: _____

June 30, 1976

[Note: 0 = Ø = zero]

INSTRUCTION TABLE

(ALPHABETIZED BY MNEMONIC)

MNE-MONIC	INSTRUCTION	NO. TIMES EXECUTED	M/INST	M SUBTOTAL	R/INST	R SUBTOTAL
A	ADD		4		16	
AD	ADD NORMALIZED (long)		8		32	
ADR	ADD NORMALIZED (long)		0		24	
AE	ADD NORMALIZED (short)		4		16	
AER	ADD NORMALIZED (short)		0		12	
AH	ADD HALFWORD		2		12	
AL	ADD LOGICAL		4		16	
ALR	ADD LOGICAL		0		12	
AP	ADD DECIMAL					
AR	ADD		0		12	
AU	ADD UNNORMALIZED (short)					
AUR	ADD UNNORMALIZED (short)					
AW	ADD UNNORMALIZED (long)					
AWR	ADD UNNORMALIZED (long)					
AXR	ADD NORMALIZED (extended)					
BAL	BRANCH AND LINK		Ø		8	
BALR	BRANCH AND LINK		Ø		15	
$BALR_0$	BRANCH AND LINK		0		9	
BC	BRANCH ON CONDITION		Ø		Ø	
BCR	BRANCH ON CONDITION		Ø		8	
BCT	BRANCH ON COUNT		Ø		2	
BCTR	BRANCH ON COUNT					
$BCTR_0$	BRANCH ON COUNT		Ø		2	
BXH	BRANCH ON INDEX HIGH					
BXLE	BRANCH ON INDEX LOW OR EQUAL		Ø		17	
C	COMPARE		4		12	
CD	COMPARE (long)					
CDR	COMPARE (long)					
CDS	COMPARE DOUBLE AND SWAP					
CE	COMPARE (short)					
CER	COMPARE (short)					
CH	COMPARE HALFWORD		2		8	
CL	COMPARE LOGICAL		4		12	
CLC *	COMPARE LOGICAL (character)					
CLCL *	COMPARE LOGICAL LONG					
CLI	COMPARE LOGICAL (immediate)					
CLM	COMPARE LOGICAL CHARACTERS UNDER MASK					
	Page Subtotals	C_{P1}:		M_{P1}:		R_{P1}:

June 30, 1976

IBM S/370 CALCULATION SHEET
Test Program: _____
Programmer: _____
Date: _____

MNEMONIC	INSTRUCTION	NO. TIMES EXECUTED	M/INST	M SUBTOTAL	R/INST	R SUBTOTAL
CLR	COMPARE LOGICAL		Ø		8	
CLRIO	CLEAR I/O					
CP	COMPARE DECIMAL					
CR	COMPARE		Ø		8	
CS	COMPARE AND SWAP					
CVB	CONVERT TO BINARY					
CVD	CONVERT TO DECIMAL		8		200	
D	DIVIDE		4		24	
DD	DIVIDE (long)		8		32	
DDR	DIVIDE (long)		0		40	
DE	DIVIDE (short)		4		16	
DER	DIVIDE (short)		0		20	
DP	DIVIDE DECIMAL					
DR	DIVIDE		Ø		2Ø	
ED	EDIT					
EDMK	EDIT AND MARK					
EX	EXECUTE		Ø		1	
HDR	HALVE (long)		0		16	
HDV	HALT DEVICE					
HER	HALVE (short)		0		8	
HIO	HALT I/O					
IC	INSERT CHARACTER		1		1	
ICM	INSERT CHARACTERS UNDER MASK					
IPK	INSERT PSW KEY					
ISK	INSERT STORAGE KEY					
L	LOAD		4		4	
LA	LOAD ADDRESS		Ø		7	
LCDR	LOAD COMPLEMENT (long)		0		16	
LCER	LOAD COMPLEMENT (short)		0		8	
LCR	LOAD COMPLEMENT		Ø		8	
LCTL*	LOAD CONTROL					
LD	LOAD (long)		8		8	
LDR	LOAD (long)		0		16	
LE	LOAD (short)		4		4	
LER	LOAD (short)		0		8	
	Page Subtotals	C_{P2}:		M_{P2}		R_{P2}

June 30, 1976

IBM S/370 CALCULATION SHEET
Test Program: _____
Programmer: _____
Date: _____

MNE-MONIC	INSTRUCTION	NO. TIMES EXECUTED	M/INST	M SUBTOTAL	R/INST	R SUBTOTAL
LH	LOAD HALFWORD		2		4	
LM*	LOAD MULTIPLE		Ø		Ø	
LNDR	LOAD NEGATIVE (long)					
LNER	LOAD NEGATIVE (short)					
LNR	LOAD NEGATIVE		Ø		8	
LPDR	LOAD POSITIVE (long)					
LPER	LOAD POSITIVE (short)					
LPR	LOAD POSITIVE		Ø		8	
LPSW	LOAD PSW		8		8	
LR	LOAD		Ø		8	
LRA	LOAD REAL ADDRESS		0		[illegible]	
LRDR	LOAD ROUNDED (extended to long)					
LRER	LOAD ROUNDED (long to short)					
LTDR	LOAD AND TEST (long)					
LTER	LOAD AND TEST (short)					
LTR	LOAD AND TEST		Ø		8	
M	MULTIPLY		4		2Ø	
MC	MONITOR CALL					
MD	MULTIPLY (long)		8		32	
MDR	MULTIPLY (long)		0		24	
ME	MULTIPLY (short to long)		4		20	
MER	MULTIPLY (short to long)		0		16	
MH	MULTIPLY HALFWORD		2		12	
MP	MULTIPLY DECIMAL					
MR	MULTIPLY		Ø		16	
MVC*	MOVE (character)					
MVCL*	MOVE LONG					
MVI	MOVE (immediate)		1		4	
MVN*	MOVE NUMERICS					
MVO	MOVE WITH OFFSET					
MVZ*	MOVE ZONES					
MXD	MULTIPLY (long to extended)					
MXDR	MULTIPLY (long to extended)					
MXR	MULTIPLY (extended)					
N	AND		4		16	
	Page Subtotals	C_{P3}:		M_{P3}:		R_{P3}:

June 30, 1976

IBM S/370 CALCULATION SHEET
Test Program: _____
Programmer: _____
Date: _____

MNE-MONICS	INSTRUCTION	NO. TIMES EXECUTED	M/INST	M SUBTOTAL	R/INST	R SUBTOTAL
NC*	AND (character)					
NI	AND (immediate)		2		3	
NR	AND		0		12	
O	OR		4		16	
OC*	OR (character)					
OI	OR (immediate)		2		3	
OR	OR		0		12	
PACK*	PACK					
PTLB	PURGE TLB					
RDD	READ DIRECT		1		1	
RRB	RESET REFERENCE BIT					
S	SUBTRACT		4		16	
SCK	SET CLOCK					
SCK C	SET CLOCK COMPARATOR					
SD	SUBTRACT NORMALIZED (long)		8		32	
SDR	SUBTRACT NORMALIZED (long)		0		24	
SE	SUBTRACT NORMALIZED (short)		4		16	
SER	SUBTRACT NORMALIZED (short)		0		12	
SH	SUBTRACT HALFWORD		2		12	
SIGP	SIGNAL PROCESSOR					
SIO	START I/O					
SIOF	START I/O FAST RELEASE					
SL	SUBTRACT LOGICAL		4		16	
SLA	SHIFT LEFT SINGLE		Ø		8	
SLDA	SHIFT LEFT DOUBLE		Ø		16	
SLDL	SHIFT LEFT DOUBLE LOGICAL		Ø		16	
SLL	SHIFT LEFT SINGLE LOGICAL		Ø		8	
SLR	SUBTRACT LOGICAL		0		12	
SP	SUBTRACT DECIMAL					
SPKA	SET PSW KEY FROM ADDRESS					
SPM	SET PROGRAM MASK		Ø		2	
SPT	SET CPU TIMER					
SPX	SET PREFIX					
SR	SUBTRACT		0		12	
SRA	SHIFT RIGHT SINGLE		Ø		8	
SRDA	SHIFT RIGHT DOUBLE		Ø		16	
SRDL	SHIFT RIGHT DOUBLE LOGICAL		Ø		16	
SRL	SHIFT RIGHT SINGLE LOGICAL		Ø		8	
SRP	SHIFT AND ROUND DECIMAL					
SSK	SET STORAGE KEY					
	Page Subtotals	C_{P4}:		M_{P4}:		R_{P4}:

June 30, 1976

IBM S/370 CALCULATION SHEET
Test Program: _____
Programmer: _____
Date: _____

MNE-MONIC	INSTRUCTION	NO. TIMES EXECUTED	M/INST	M SUBTOTAL	R/INST	R SUBTOTAL
SSM	SET STYSTEM MASK		1		1	
ST	STORE		4		4	
STAP	STORE CPU ADDRESS					
STC	STORE CHARACTER		1		1	
STCK	STORE CLOCK					
STCK C	STORE CLOCK COMPARATOR					
STCM	STORE CHARACTERS UNDER MASK					
STCTL*	STORE CONTROL					
STD	STORE (long)		8		8	
STE	STORE (short)		4		4	
STH	STORE HALFWORD		2		2	
STIDC	STORE CHANNEL ID					
STIDP	STORE CPU ID					
STM*	STORE MULTIPLE		ø		ø	
STNSM	STORE THEN AND SYSTEM MASK					
STOSM	STORE THEN OR SYSTEM MASK					
STPT	STORE CPU TIMER					
STPX	STORE PREFIX					
SU	SUBTRACT UNNORMALIZED (short)					
SUR	SUBTRACT UNNORMALIZED (short)					
SVC	SUPERVISOR CALL		ø		20	
SW	SUBTRACT UNNORMALIZED (long)					
SWR	SUBTRACT UNNORMALIZED (long)					
SXR	SUBTRACT NORMALIZED (extended)					
TCH	TEST CHANNEL					
TIO	TEXT I/O					
TM	TEST UNDER MASK		1		3	
TR*	TRANSLATE					
TRT*	TRANSLATE AND TEST					
TS	TEST AND SET					
UNPK	UNPACK					
WRD	WRITE DIRECT					
X	EXCLUSIVE OR		4		16	
XC*	EXCLUSIVE OR (character)					
XI	EXCLUSIVE OR (immediate)		2		3	
XR	EXCLUSIVE OR		0		12	
ZAP	ZERO AND ADD					
	Page Subtotals	C_{P5}:		M_{P5}:		R_{P5}:

June 30, 1976

IBM S/370 CALCULATION SHEET
Test Program: _____
Programmer: _____
Date: _____

MISCELLANEOUS M AND R TABLE

INSTRUCTION MNEMONIC	WHAT TO COUNT	COUNT	M/COUNT	M SUBTOTAL	R/COUNT	R SUBTOTALS
CLC	Characters compared		2		22	
CLCL	Characters compared		2		28	
LM	Registers loaded		4		11	
LCTL	Registers loaded		4		11	
MVC	Characters moved		2		20	
MVCL	Characters moved					
MVN	characters moved		3		22	
MVZ	characters moved		3		22	
NC	Length of character operands		3		22	
OC	Length of character operands		3		22	
PACK	length of second operand (L2)		$2+\frac{3(L2)}{2}$ [truncate result]		$30(1+\frac{L2}{2})$ [trunciate result]	
STCTL	registers stored		4		11	
STM	Registers stored		4		11	
TR	Characters translated		3		28	
TRT	Characters compared		2		20	
XC	Length of character operands		3		22	
	Subtotals	X	X	I_M:	X	R_M:

July 7, 1976
D.A. Lamb

Instructions for Completing the DEC PDP-11 Calculation Sheet

To assist you in the computation of the S, M, and R measures for the DEC PDP-11, we have included here a set of **calculation sheets** (8 pages) and a liberal supply of **work sheets**. The work sheets are designed to be directly attached to the listing of your test program and have entries for each line of your program. Use as many work sheets as you have pages of code in your listing. The calculation sheets will be used to enumerate S, M, and R measures via addressing modes and instruction type.

The principle use of the work sheets will be to get accurate counts of instruction executions. The calculation sheets will be used to determine the final S, M, and R measures. Later we will indicate how to obtain these measures from your work sheets directly, but this will be used primarily as a check on your calculations.

1. You will need to refer to most of the calculation sheets at the same time, so you should work at a desk with enough space to spread out all of the sheets at once. You can lay aside the summary calculation sheet (page 1) since you will not need it until the end. You will also need the CFA Test Data Specifications of June 16, 1976.

2. Attach the work sheets to your program. The work sheets have a line for each line of your program, and a column for the instruction type (Mode), the number of times the program is executed (N), the storage measure (S), the memory transfer measure (M), the register transfer measure (R), and the products N*M and N*R.

3. Go through your program to determine how often each instruction will be executed, using the test data items marked with a plus (+) in the specifications. Fill in the N column of the work sheet with these numbers. Where there is more than one set of test data for a particular program, determine the number of times each instruction is executed for each set of data, add the results together, and write the sum in the N column. If your program is simple enough, you may be able to do an analysis to determine these numbers in terms of the size of the test data and some simple characteristics (such as the number of zeros in the matrix for the boolean matrix transpose program). Failing that, you may need to hand-simulate the operation of your program on the test data.

4. Perform steps 5 to 11 for each instruction in the program. Each step is labeled with the name of the sheet on which the number calculated in the step is to be written.

5. work sheet: Determine the type of the instruction (d=double operand, s=single operand, r=register+operand, f=floating point, m=miscellaneous) and mark the type in the Mode column of the work sheet.

6. calculation sheet: Find the S measure table on page 2. Add one to the "count"

column of the "instruction" row. For each immediate or absolute operand (modes 2 and 3 using the PC) add one to the "count" column of the "immediate and absolute" row. For each indexed and indexed deferred operand, add one to the "indexed operands" row.

7. work sheet: For each "one" added to the S measure table in step 6, count two, and add the resulting number to the S column of the work sheet. This gives the number of bytes used for the instruction.

8. calculation sheet: For each operand in the instruction, find the appropriate row in the Address Mode table (page 2). Note that there are separate entries for byte and word instructions for modes 2 and 4 (autoincrement and autodecrement). Add one to the "no. of times executed" column of the row.

10. calculation sheet: Find the row of the Instruction Table (pages 3 to 10) corresponding to the instruction. The instructions are grouped into tables based on the type determined in step 5. Those instructions which have 6-bit operand fields have two or three sections, one labeled "basic", and the others labeled "source mode > 0", "dest mode > 0", or "address mode > 0". Add the number in the N column of the work sheet to the "basic" field of the "no. of times executed" column of this row. For double operand instructions, add N to the "source mode > 0" entry if the addressing mode of the source operand is not register mode (mode 0), and add N to the "dest mode > 0" entry if the addressing mode of the destination operand is not register mode. The single operand, register operand, and floating point instructions should be treated similarly. If the "R/INST" and "M/INST" columns of this row are blank, mark the row (preferably in a colour that stands out), write zeros in these two columns, and on a separate sheet make note of the fact that the instruction was missing.

10. work sheet: Add the number in the "R/INST" column of the row found in step 9 and the number in the "R/MODE" column of the row found in step 8. Write this sum in the R column of the work sheet.

11. work sheet: Add the number in the "M/INST" column of the row found in step 9 and the number in the "M/MODE" column of the row found in step 8. Write this sum in the M column of the work sheet.

When you have completed these calculations for the entire program, compute the totals in the tables as outlined in steps 12 to 18.

12. calculation sheet: For each entry of the instruction tables (page 3 through 10), multiply the number in the "no. of times executed" column by the number in the "M/INST" column and write the product in the "M subtotal" column. Multiply the number in the "no. of times executed" column by the number in the "R/INST" column and write the product in the "R subtotal" column. Treat the "basic" and "mode > 0" entries for each instruction as separate entries for the purposes of this calculation.

13. calculation sheet: For each page of the instruction tables, calculate the subtotals

of the "number of times executed", "M subtotal", and "R subtotal" columns, fill in the subtotals at the bottom of the page, and write these subtotals in the space provided on the Summary Calculation page (page 1).

14. calculation sheet: On the S Measure Table (page 2), add up the "S subtotal" column to give S_I. Fill in the corresponding entry on the Summary Calculation page.

15. calculation sheet: On the Address Mode Table, add up the columns giving C_{am}, M_{am}, and R_{am}, and copy these to the summary page.

16. calculation sheet: Add up the entries on the summary page to give the S, M, and R measures.

17. calculation sheet: Perform the consistency check given on the summary page.

18. work sheet: Multiply the N column of the work sheet by the M column and write the result in the NM column. Multiply the N column by the R column and write the result in the NR column. Calculate the sum of the NM column; this should be the same as the final M measure. Similarly, the sum of the NR column should be the same as the final R measure.

PDP-11 CALCULATION SHEET
S, M, AND R MEASURES
SUMMARY CALCULATIONS

July 2, 1976

Test Program: _____
Programmer: _____
Date: _____

S MEASURE

Instruction subtotal, S_i _____

Constants and local variables: _____

Dynamic workspace (includes stack area): _____

S [ ]

M MEASURE

Instruction subtotals, M_d : _____

M_r : _____

M_{s1}: _____

M_{s2}: _____

M_{misc}: _____

M_{f1}: _____

M_{f2}: _____

M_{f3}: _____

Addressing mode subtotal, M_{am}: _____

M [ ]

R MEASURE

Instruction subtotals, R_d : _____

R_r : _____

R_{s1}: _____

R_{s2}: _____

R_{misc}: _____

R_{f1}: _____

R_{f2}: _____

R_{f3}: _____

Addressing mode subtotal, R_{am}: _____

R [ ]

CONSISTENCY CHECKS ON CALCULATIONS

You can use the following identity to provide some measure of assurance you have not made a clerical error in tabulating the S, M, and R measures:

$$(1)\quad C_{f1} + C_{f2} + C_{f3} + 2C_{d0} + C_{r0} + C_{s1} + C_{s2} = C_{am}$$

ADDRESS MODE TABLE

ADDRESSING MODE	NO. TIMES EXECUTED	M/MODE	M Subtotal	R/MODE	R SUBTOTAL
ø	C_{amo}:	ø		ø	
1		ø		4	
2 (byte mode)		ø		6	
2 (word mode)		ø		7	
3		2		9	
4 (byte mode)		ø		6	
4 (word mode)		ø		7	
5		2		9	
6		2		15	
7		4		17	
	C_{am}:		M_{am}:		R_{am}:

S Measure Table

Item	Count	S/COUNT	S Subtotal
Instructions		2	
Immediate and Absolute Operands (# and @#)		2	
Indexed Operands (modes 6 and 7)		2	
			S_I

Floating Point Instruction Table - Page 1

Mnemonic	Instruction		No. Times Executed	M/INST	M Sub-total	R/INST	R Sub-total
ABSD	Make absolute (double)	basic		2		11	
		mode > 0		2		0	
ABSF	Make Absolute (floating)	basic		2		11	
		mode > 0		2		0	
ADDD	Add (double)	basic		2		33	
		mode > 0		8		8	
ADDF	Add (floating)	basic		2		21	
		mode > 0		4		4	
CFCC	Copy floating condition codes			2		11	
CLRD	Clear (double)	basic		2		17	
		mode > 0		8		-8	
CLRF	Clear (floating)	basic		2		13	
		mode > 0		4		-4	
CMPD	Compare (double)	basic		2		25	
		mode > 0		8		8	
CMPF	Compare (floating)	basic		2		17	
		mode > 0		4		4	
DIVD	Divide (double)	basic		2		33	
		mode > 0		8		8	
DIVF	Divide (floating)	basic		2		21	
		mode > 0		4		4	
LDCDF	Load, converting double to floating	basic		2		21	
		mode > 0		8		8	
LDCFD	Load, converting floating to double	basic		2		21	
		mode > 0		4		4	
LDCID	Load, converting short to double	basic		2		27	
		mode > 0		2		-2	
LDCIF	Load, converting short to floating	basic		2		23	
		mode > 0		2		-2	
LDCLD	Load, converting long to double	basic		2		27	
		mode > 0		4		-2	
LDCLF	Load, converting long to floating	basic		2		23	
		mode > 0		4		-2	
	Page subtotals		C_{F1}		M_{F1}		R_{F1}

Floating Point Instruction Table - Page 2

Mnemonic	Instruction		No. Times Executed	M/INST	M Subtotal	R/INST	R Subtotal
LDD •	Load (double)	basic		2		25	
		mode > 0		8		- 8	
LDEXP	Load Exponent	basic		2		12	
		mode > 0		1		1	
LDF	Load (floating)	basic		2		17	
		mode > 0		4		- 4	
LDFPS	Load floating status	basic		2		13	
		mode > 0		2		- 2	
MODD	Multiply and integrize (double)	basic					
		mode > 0					
MODF	Multiply and integrize (floating)	basic					
		mode > 0					
MULD	Multiply (double)	basic		2		41	
		mode > 0		8		8	
MULF	Multiply (floating)	basic		2		21	
		mode > 0		4		4	
NEGD	Negate (double)	basic		2		11	
		mode > 0		2		0	---
NEGF	Negate (floating)	basic		2		11	
		mode > 0		2		0	---
SETD	Set double mode			2		10	
SETF	Set Floating mode			2		10	
SETI	Set short integer mode			2		10	
SETL	Set long integer mode			2		10	
STCDF	Store, converting double to floating	basic		2		21	
		mode > 0		4		- 4	
STCDI	Store, converting double to integer	basic		2		19	
		mode > 0		2		- 2	
STCDL	Store, converting double to long	basic		2		19	
		mode > 0		4		- 2	
STCFD	Store, converting floating to double	basic		2		21	
		mode > 0		8		- 8	
	Page subtotals		C_{F2}		M_{F2}		R_{F2}

Floating Point Instruction Table - Page 3

Mnemonic	Instruction		No. Times Executed	M/INST	M Subtotal	R/INST	R Subtotal
STCFI	Store, converting floating to integer	basic		2		15	
		mode > 0		2		- 2	
STCFL	Store, converting floating to long	basic		2		17	
		mode > 0		4		- 4	
STD	Store (double)	basic		2		25	
		mode > 0		8		- 8	
STEXP	Store Exponent	basic		2		12	
		mode > 0		1		- 1	
STF	Store (floating)	basic		2		17	
		mode > 0		4		- 4	
STFPS	Store floating status	basic		2		13	
		mode > 0		2		- 2	
STST	Store status and exception address	basic		2		13	
		mode > 0		4		0	
SUBD	Subtract (double)	basic		2		33	
		mode > 0		8		8	
SUBF	Subtract (floating)	basic		2		21	
		mode > 0		4		4	
TSTD	Test (double)	basic		2		11	
		mode > 0		2		2	
TSTF	Test (floating)	basic		2		11	
		mode > 0		2		2	
	Page subtotals		C_{F3}		M_{F3}		R_{F3}

Single Operand Instruction Table - Page 1

Mnemonic	Instruction		No. Times Executed	M/INST	M Subtotal	R/INST	R Subtotal
ADC	Add carry	basic		2		13	
		mode > 0		4		0	---
ADCB	Add carry byte	basic		2		11	
		mode > 0		2		0	---
ASL	Arithmetic Shift Left	basic		2		13	
		mode > 0		4		0	---
ASLB	Arithmetic Shift Left Byte	basic		2		11	
		mode > 0		2		0	---
ASR	Arithmetic Shift Right	basic		2		13	
		mode > 0		4		0	---
ASRB	Arithmetic Shift Right Byte	basic		2		11	
		mode > 0		2		0	---
CLR	Clear	basic		2		11	
		mode > 0		2		- 2	
CLRB	Clear Byte	basic		2		10	
		mode > 0		1		- 1	
COM	Complement	basic		2		13	
		mode > 0		4		0	---
COMB	Complement Byte	basic		2		11	
		mode > 0		2		0	---
DEC	Decrement	basic		2		11	
		mode > 0		4		1	
DECB	Decrement Byte	basic		2		11	
		mode > 0		2		0	---
INC	Increment	basic		2		11	
		mode > 0		4		1	
INCB	Increment Byte	basic		2		11	
		mode > 0		2		0	---
JMP	Jump			2		13	
JSR	Jump to Subroutine			4		26	
NEG	Negate	basic		2		13	
		mode > 0		4		0	---
	Page subtotals		C_{S1} $C_{S.am1}$		M_{S1}		R_{S1}

Single Operand Instruction Table - Page 2

Mnemonic	Instruction		No. Times Executed	M/INST	M Sub-total	R/INST	R Sub-total
NEGB	Negate Byte	basic		2		11	
		mode > 0		2		0	---
ROL	Rotate Left	basic		2		13	
		mode > 0		4		0	---
ROLB	Rotate Left Byte	basic		2		11	
		mode > 0		2		0	---
ROR	Rotate Right	basic		2		13	
		mode > 0		4		0	---
RORB	Rotate Right Byte	basic		2		11	
		mode > 0		2		0	---
SBC	Subtract carry	basic		2		13	
		mode > 0		4		0	---
SBCB	Subtract carry byte	basic		2		11	
		mode > 0		2		0	---
SWAB	Swap Bytes	basic		2		14	
		mode > 0		4		0	---
SXT	Sign Extend	basic		2		11	
		mode > 0		2		-2	
TST	Test	basic		2		11	
		mode > 0		2		2	
TSTB	Test Byte	basic		2		10	
		mode > 0		1		1	
	Page subtotals		C_{S2}	$C_{S.am2}$	M_{S2}		R_{S2}

Miscellaneous Instruction Table

Mnemonic	Instruction	No. Times Executed	M/INST	M Subtotal	R/INST	R Subtotal
BPT	Breakpoint Trap		6		33	
	Branches		2		14	
EMT	Emulator Trap		6		33	
IOT	I/O Trap		6		33	
RTI	Return from Interrupt		6		27	
RTS	Return from Subroutine		4		22	
SOB	Subtract One and Branch		2		17	
TRAP			6		33	
	Page subtotals	C_{misc}		M_{misc}		R_{misc}

DOUBLE OPERAND INSTRUCTION TABLE July 2, 1976

(Alphabetized by Mnemonic)

MNE-MONIC	INSTRUCTION		NO TIMES EXECUTED BASIC	NO TIMES EXECUTED ADDRESS MODE	M/ INST	M Sub-Total	R/ Inst	R Sub-Total
ADD	add	basic count			2		15	
		source mode > 0			2		2	
		dest. mode > 0			4		0	
BIC	bit clear	basic count			2		15	
		source mode > 0			2		2	
		dest. mode > 0			4		0	---
BICB	bit clear byte	basic count			2		12	
		source mode > 0			1		1	
		dest. mode > 0			2		0	---
BIS	bit set (OR)	basic count			2		15	
		source mode > 0			2		2	
		dest. mode > 0			4		0	---
BISB	bit set (OR) byte	basic count			2		12	
		source mode > 0			1		1	
		dest. mode > 0			2		0	---
BIT	bit test (AND)	basic count			2		13	
		source mode > 0			2		2	
		dest. mode > 0			2		2	
BITB	bit test (AND) byte	basic count			2		11	
		source mode > 0			1		1	
		dest. mode > 0			1		1	
CMP	compare	basic count			1		13	
		source mode > 0			2		2	
		dest. mode > 0			2		2	
COMPB	compare byte	basic count			2		11	
		source mode > 0			1		1	
		dest. mode > 0			1		1	
MOV	move	basic count			2		13	
		source mode > 0			2		2	
		dest. mode > 0			2		- 2	
MOVB	move byte	basic count			2		11	
		source mode > 0			1		1	
		dest. mode > 0			1		- 1	
SUB	subtract	basic count			2		15	
		source mode > 0			2		2	
		dest. mode > 0			4		0	
		Subtotals	C_{d0}	$C_{d.am}$		M_d		R_d

DOUBLE OPERAND INSTRUCTION TABLE

(Alphabetized by Mnemonic)

MNE-MONIC	INSTRUCTION		NO TIMES EXECUTED BASIC	NO TIMES EXECUTED ADDRESS MODE	M/ INST	M Sub-Total	R/ Inst	R Sub-Total
ASH	shift arith-metically	basic count			2		14	
		address mode > 0			2		2	
ASHC	arith shift combined	basic count			2		18	
		address mode > 0			2		2	
DIV	divide	basic count			2		19	
		address mode > 0			2		2	
MUL	multiply	basic count			2		17	
		address mode > 0			2		2	
XOR	exclusive OR	basic count			2		15	
		address mode > 0			4		0	---
		Subtotals	C_{r0}	$C_{r.am}$		M_r:		Rr:

July 7, 1976
D.A. Lamb

Instructions for Completing the Interdata 8/32 Calculation Sheet

To assist you in the computation of the S, M, and R measures for the Interdata 8/32, we have included here a set of calculation sheets (8 pages) and a liberal supply of work sheets. The work sheets are designed to be directly attached to the listing of your test program and have entries for each line of your program. Use as many work sheets as you have pages of code in your listing. The calculation sheets will be used to enumerate S, M, and R measures via addressing modes and instruction type.

The principle use of the work sheets will be to get accurate counts of instruction executions. The calculation sheets will be used to determine the final S, M, and R measures. Later we will indicate how to obtain these measures from your work sheets directly, but this will be used primarily as a check on your calculations.

1. You will need to refer to most of the calculation sheets at the same time, so you should work at a desk with enough space to spread out all of the sheets at once. You can lay aside the summary calculation sheet (page 1) since you will not need it until the end. You will also need the CFA Test Data Specifications of June 16, 1976.

2. Attach the work sheets to your program. The work sheets have a line for each line of your program, and a column for the instruction type (Mode), the number of times the program is executed (N), the storage measure (S), the memory transfer measure (M), the register transfer measure (R), and the products N*M and N*R.

3. Go through your program to determine how often each instruction will be executed, using the test data items marked with a plus (+) in the specifications. Fill in the N column of the work sheet with these numbers. Where there is more than one set of test data for a particular program, determine the number of times each instruction is executed for each set of data, add the results together, and write the sum in the N column. If your program is simple enough, you may be able to do an analysis to determine these numbers in terms of the size of the test data and some simple characteristics (such as the number of zeros in the matrix for the boolean matrix transpose program). Failing that, you may need to hand-simulate the operation of your program on the test data.

4. Perform steps 5 to 12 for each instruction in the program. Each step is labeled with the name of the sheet on which the number calculated in the step is to be written.

5. work sheet: Determine the type of the instruction (RR, SF, RX1, RX2, RX3, RI1, or RI2) and mark the type in the Mode column of the work sheet.

6. calculation sheet: Find the row of the Instruction Format Table (page 2) corresponding to the type determined in step 5. Add one to the "no. static

occurrences" column of that row, and add the number in the N column of the work sheet to the "no. times executed" column of the row.

7. work sheet: Copy the number from the "S/INST" column of the row in step 6 to the S column of the work sheet.

8. calculation sheet: For each effective address calculation in the instruction, find the appropriate row in the Address Calculation table (page 2). RR and SF instructions are included even though they in reality have no address calculation. The addressing modes which involve index or base registers have separate rows depending on whether the register involved is register zero or not. Add one to the "no. of times executed" column of the row.

9. calculation sheet: Find the row of the Instruction Table (pages 3 to 9) corresponding to the instruction. Add the number in the N column of the work sheet to the "no. of times executed" column of this row. If the "R/INST" and "M/INST" columns of this row are blank, mark the row (preferably in a colour that stands out), write zeros in these two columns, and on a separate sheet make note of the fact that the instruction was missing.

10. work sheet: Add the number in the "R/INST" column of the row found in step 9 and the number in the "R/INST" column of the row found in step 8. Write this sum in the R column of the work sheet.

11. work sheet: Add the number in the "M/INST" column of the row found in step 9 and the number in the "M/INST" column of the row found in step 6. Write this sum in the M column of the work sheet.

12a. calculation sheet: If the entry in the Instruction Table (found in step 9) has an asterisk (*), the M and R measures depend on some other count associated with the instruction. For example, a STM (store multiple) instruction costs 4 bytes in the M measure for every register stored. For each such starred instruction, look up its entry in the "Miscellaneous M and R table" (page 9), calculate the appropriate "what to count" number, multiply by the entry in the N column of the work sheet, and add this number to the "count" column of the table.

12b. work sheet: Multiply the "what to count number" determined in step 12a by the number in the "M/COUNT" entry used in that step, and add this product to the M column of the work sheet.

When you have completed these calculations for the entire program, compute the totals in the tables as outlined in steps 13 to 19.

13. calculation sheet: For each entry of the Instruction Table (page 3 through 9), multiply the number in the "no. of times executed" column by the number in the "M/INST" column and write the product in the "M subtotal" column. Multiply the number in the "no. of times executed" column by the number in the "R/INST" column and write the product in the "R subtotal" column.

14. calculation sheet: For each page of the Instruction table, calculate the subtotals of

the "number of times executed", "M subtotal", and "R subtotal" columns, fill in the subtotals at the bottom of the page (C_{P1}, M_{P1}, R_{P1} for instruction page 1, and so on), and write these subtotals in the space provided on the Summary Calculation page (page 1).

15. calculation sheet: On the Instruction Format Table (page 2), add up the "S subtotals", "number of times executed", and "M subtotal" columns to give S_I, C_I, and M_I. Fill in the corresponding entries on the Summary Calculation page.

16. calculation sheet: On the Effective address Calculation table, add up the columns giving C_{eac} and R_{eac}, and copy these to the summary page.

17. calculation sheet: Add up the entries on the summary page to give the S, M, and R measures.

18. calculation sheet: Perform the consistency checks given on the summary page. You should get C_I equal to C_{P1} + ... + C_{P7}, and C_{eac} equal to $C_{RR,SF}$ + $C_{RX1,RX2,RI1}$ + $C_{RX3,RI2}$.

19. work sheet: Multiply the N column of the work sheet by the M column and write the result in the NM column. Multiply the N column by the R column and write the result in the NR column. Calculate the sum of the NM column; this should be the same as the final M measure. Similarly, the sum of the NR column should be the same as the final R measure.

Interdata 8/32 Calculation Sheet

S, M, and R Measures

Summary Calculations

6 July 1976

Test program:_____

Programmer:_____

Date:_____

S Measure

Instruction Subtotal, S_I: _____

Constants and Local Variables _____

Temporary Workspace: _____

S [ ]

M Measure

Instruction Format Subtotal, M_I: _____

Instruction Page Subtotal, M_{P1}: _____

M_{P2}: _____

M_{P3}: _____

M_{P4}: _____

M_{P5}: _____

M_{P6}: _____

M_{P7}: _____

Miscellaneous Subtotal, M_m: _____

M [ ]

R Measure

Instruction Format Subtotal, R_I: _____

Instruction Page Subtotal, R_{P1}: _____

R_{P2}: _____

R_{P3}: _____

R_{P4}: _____

R_{P5}: _____

R_{P6}: _____

R_{P7}: _____

Miscellaneous Subtotal, R_m: _____

Address Calculation Subtotal, R_{eac}: _____

R [ ]

Consistency Checks on Calculations

You can use the following identities to provide some measure of assurance you have not made a clerical error in tabulating the S, M and R measures:

(1) $C_{P1} + C_{P2} + C_{P3} + C_{P4} + C_{P5} + C_{P6} + C_{P7} = C_I$

(2) $C_{RR} + C_{RX1} + C_{RX3} = C_{eac}$

Address Calculation Table - Interdata 8/32

Instruction Format	Type of Address Calculation	No. Times Executed	R/INST	R subtotal
RR			8	
SF			8	
RX1	X2=0		21	
RX1	X2≠0		24	
RX2	X2=0		24	
RX2	X2≠0		33	
RX3	FX1=0,SX1=0		27	
RX3	FX1=0,SX1≠0		30	
RX3	FX1≠0,SX1=0		30	
RX3	FX1≠0,SX1≠0		39	
RI1	X2=0		19	
RI1	X2≠0		23	
RI2	X2=0		26	
RI2	X2≠0		30	
Address Calculation Subtotal		C_{eac}		R_{eac}

Instruction Format Table

Instruction Format	No. Static Occurences	S/INST	S subtotals	No. Times Executed	M/INST	M subtotals
RR, SF		2		C_{RR}:	2	
RX1, RX2, RI1		4		C_{RX1}:	4	
RX3, RI2		6		C_{RX3}:	6	
Instruction Format Subtotal			S_I	C_I		M_I

Instruction Table - Page 1

Mnemonic	Instruction	No. Times Executed	M/INST	M subtotal	R/INST	R subtotal
A	Add		4		16	
ABL	Add to Bottom of List		16		38	
AE	Add floating point		4		16	
AER	Add floating point Register		0		12	
AH	Add Halfword		2		12	
AHI	Add Halfword Immediate		0		10	
AHM	Add Halfword to Memory		4		8	
AI	Add Immediate		0		12	
AIS	Add Immediate Short		0		7	
AL	AutoLoad					
AM	Add to Memory		8		12	
AR	Add register		0		12	
ATL	Add to Top of List		16		38	
BAL	Branch and Link		0		3	
BALR	Branch and Link Register		0		20	
BDCS	Branch to Control Store					
BFBS	Branch on False Condition Backward Short		0		1	
BFC	Branch on False Condition		0*		-1	
BFCR	Branch on False Condition Register		0		6	
BFFS	Branch on False Condition Forward Short		0		1	
BTBS	Branch on True Condition Backward Short		0		1	
BTC	Branch on True Condition		0		-1	
BTCR	Branch on True Condition Register		0		6	
BTFS	Branch on True Condition Forward Short		0		1	
BXH	Branch on Index High		0		17	
	Page subtotals	C_{P1}		M_{P1}		R_{P1}

Instruction Table - Page 2

Mnemonic	Instruction	No. Times Executed	M/INST	M subtotal	R/INST	R subtotal
BXLE	Branch on Index Low or Equal		0		17	
C	Compare		4		12	
CBT	Complement Bit		4		20	
CE	Compare Floating Point		4		12	
CER	Compare Floating Point Register		0		8	
CH	Compare Halfword		2		8	
CHI	Compare Halfword Immediate		0		6	
CHVR	Convert to Halfword Value Register					
CI	Compare Immediate		0		8	
CL	Compare Logical		4		12	
CLB	Compare Logical Byte		1		3	
CLH	Compare Logical Halfword		2		8	
CLHI	Compare Logical Halfword Immediate		0		6	
CLI	Compare Logical Immediate		0		8	
CLR	Comare Logical Register		0		8	
CR	Comare Register		0		8	
CRC12	Cyclic Redundancy Check modulo 12					
CRC16	Cyclic Redundancy Check modulo 16					
D	Divide		4		24	
DE	Divide Floating Point		4		16	
DER	Divide Floating Point Register		0		12	
DH	Divide Halfword		2		16	
DHR	Divide Halfword Register		0		14	
DR	Divide Register		0		20	
ECS	Enter Control Store					
	Page subtotals	C_{P2}		M_{P2}		R_{P2}

Instruction Table - Page 3

Mnemonic	Instruction	No. Times Executed	M/INST	M subtotal	R/INST	R subtotal
EPSR	Exchange Program Status Register		0		16	
EXBR	Exchange Byte Register		0		4	
EXHR	Exchange Halfword Register		0		8	
FLR	Float Register		0		16	
FXR	Fix Register		0		8	
L	Load		4		4	
LA	Load Address		0		7	
LB	Load Byte		1		1	
LBR	Load Byte Register		0		2	
LCS	Load Complement Short		0		5	
LE	Load Floating Point		4		4	
LER	Load Floating Point Register		0		8	
LH	Load Halfword		2		4	
LHI	Load Halfword Immediate		0		8	
LHL	Load Halfword Logical		2		4	
LI	Load Immediate		0		8	
LIS	Load Immediate Short		0		5	
LM *	Load Multiple					
LME	Load Floating Point Multiple					
LPSW	Load Program Status Word		8		8	
LPSWR	Load Program Status Word Register		0		16	
LR	Load Register		0		8	
M	Multiply		4		20	
ME	Multiply Floating Point		4		16	
MER	Multiply Floating Point Register		0		12	
	Page subtotals	C_{P3}		M_{P3}		R_{P3}

Instruction Table - Page 4

Mnemonic	Instruction	No. Times Executed	M/INST	M subtotal	R/INST	R subtotal
MH	Multiply Halfword		2		10	
MHR	Multiply Halfword Register		0		12	
MR	Multiply Register		0		16	
N	AND		4		16	
NH	AND Halfword		2		12	
NHI	AND Halfword Immediate		0		10	
NI	AND Immediate		0		12	
NR	AND Register		0		12	
O	OR		4		16	
OC	Output Command		1		4	
OCR	Output Command Register		0		3	
OH	OR Halfword		2		12	
OHI	OR Halfword Immediate		0		10	
OI	OR Immediate		0		12	
OR	OR Register		0		12	
RB	Read Block					
RBL	Remove from Bottom of List		16		38	
RBR	Read Block Register					
RBT	Reset Bit		4		20	
RD	Read Data					
RDCS	Read Control Store					
RDR	Read Data Register					
RH	Read Halfword					
RHR	Read Halfword Register					
RLL			0		8	
	Page subtotals	C_{P4}		M_{P4}		R_{P4}

Instruction Table - Page 5

Mnemonic	Instruction	No. Times Executed	M/INST	M subtotal	R/INST	R subtotal
RRL			0		8	
RTL	Remove from Top of List		16		38	
S	Subtract		4		16	
SBT	Set Bit		4		20	
SCP	Simulate Channel Program					
SE	Subtract Floating Point		4		16	
SER	Subtract Floating Point Register		0		12	
SH	Subtract Halfword		2		12	
SHI	Subtract Halfword Immediate		0		10	
SI	Subtract Immediate		0		12	
SINT	Simulate Interrupt					
SIS	Subtract Immediate Short		0		7	
SLA	Shift Left Arithmetic		0		8	
SLHA	Shift Left Halfword Arithmetic		0		4	
SLHL	Shift Left Halfword Logical		0		4	
SLHLS	Shift Left Halfword Logical Short		0		6	
SLL	Shift Left Logical		0		8	
SLLS	Shift Left Logical Short		0		10	
SR	Subtract Register		0		12	
SRA	Shift Right Arithmetic		0		8	
SRHA	Shift Right Halfword Arithmetic		0		4	
SRHL	Shift Right Halfword Logical		0		4	
SRHLS	Shift Right Halfword Logical Short		0		6	
SRL	Shift Right Logical		0		8	
SRLS	Shift Right Logical Short		0		10	
	Page subtotals	C_{P5}		M_{P5}		R_{P5}

Instruction Table - Page 6

Mnemonic	Instruction	No. Times Executed	M/INST	M subtotal	R/INST	R subtotal
SS	Sense Status		1		4	
SSR	Sense Status Register		0		3	
ST	Store		4		4	
STB	Store Byte		1		1	
STBR	Store Byte Register		0		2	
STE	Store Floating Point		4		4	
STH	Store Halfword		2		2	
STM *	Store Multiple					
STME	Store Multiple Floating Point					
SVC	Supervisor Call					
TBT	Test Bit		2		8	
THI	Test Halfword Immediate					
TI	Test Immediate					
TLATE	Translate (branch)		2		10	
TLATE	Translate (no branch)		2		15	
TS	Test and Set		2		5	
WB	Write Block					
WBR	Write Block Register					
WD	Write Data					
WDCS	Write Control Store					
WDR	Write Data Register		0		3	
WH	Write Halfword		2		6	
WHR	Write Halfword Register		0		4	
X	Exclusive OR		4		16	
XH	Exclusive OR Halfword		2		12	
	Page subtotals	C_{P6}		M_{P6}		R_{P6}

Instruction Table - Page 7

Mne-monic	Instruction	No. Times Executed	M/INST	M subtotal	R/INST	R subtotal
XHI	Exclusive OR Halfword Immediate		0		10	
XI	Exclusive OR Immediate		0		12	
XR	Exclusive OR Register	•	0		12	
	Page subtotals	C_{P7}		M_{P7}		R_{P7}

Miscellaneous M and R Table

Mne-monic	What to Count	count	M/Count	M subtotal	R/count	R subtotal
LM	registers loaded		4		11	
STM	register stored		4		11	
LME	registers loaded		4		11	
STME	registers stored		4		11	
Miscellaneous Subtotals				M_m		R_m

APPENDIX E - S, M, and R MEASURES FOR EACH TEST PROGRAM

INDIVIDUAL R MEASURES

Test Program	Computer Architecture		
	IBM S/370	PDP-11	Interdata 8/32
A. Priority I/O Kernel	947[3]	108[4]	166[12]
	2146[12]	106[12]	166[17]
	3052[14]	106[14]	214[14]
B. FIFO I/O Kernel	2222[2]	1096[2]	698[2]
	4583[13]	810[3]	937[4]
	2226[17]	1419[13]	482[13]
C. I/I Device Handler	1789[1]	1480[1]	1902[1]
	1729[17]	1416[17]	1391[17]
D. Large FFT	62904[11]	70512[11]	60446[11]
	62904[9]*	70512[9]*	50045[9]*
E. Character Search	5603[1]	4348[1]	5885[1]
	5549[4]	4326[11]	3139[3]
	10239[11]	3091[17]	5767[11]
F. Bit Test, Set, Reset	1674[9]	832[3]	891[4]
	1542[12]	917[9]	887[9]
	1212[17]	801[12]	1167[12]
			1281[11]A
G. Runge-Kutta Int.	845966[2]	724372[2]	696085[2]
	1203952[17]	665529[3]	696049[4]
		1012727[17]	777846[11]A
			874923[17]
H. Linked List Insertion	950[4]	1025[13]	834[3]
	1741[13]	1087[14]	1049[13]
	1137[14]	1210[17]	965[14]
I. Quicksort	7618[5]	74278[5]	13315[5]
	7540[6]	15205[6]	9609[6]
J. ASCII to Float-Pt.	1330[4]	1726[5]	2100[3]
	2578[5]	1512[7]	2270[5]
	2226[7]	1716[17]	1897[17]
K. Boolean Matrix	5576[3]	3180[4]	2216[6]
	5661[6]	3905[6]	3154[8]
	5277[8]	4445[8]	3945[17]
L. Virtual Memory Exchange	1931[3]	2616[4]	2539[7]
	1934[7]	2911[7]	4573[8]
	2529[8]	4226[8]	2643[17]

INDIVIDUAL M MEASURE

Test Program	Computer Architecture		
	IBM S/370	PDP-11	Interdata 8/32
A. Priority I/O Kernel	212 [3] 354 [12] 522 [14]	28 [4] 24 [12] 24 [14]	28 [12] 32 [14] 28 [17]
B. FIFO I/O Kernel	424 [2] 920 [13] 434 [17]	208 [2] 188 [3] 296 [13]	192 [2] 226 [4] 114 [13]
C. I/O Device Handler	328 [1] 304 [17]	309 [1] 290 [17]	426 [1] 279 [17]
D. Large FFT	10810 [11] 10810 [9]*	14746 [11] 14746 [9]*	10886 [11] 8560 [9]* 8560 [17]A
E. Character Search	854 [1] 940 [4] 1724 [11]	730 [1] 770 [11] 520 [17]	958 [1] 1044 [3] 1021 [11]
F. Bit Test,Set,Reset	378 [9] 358 [12] 238 [17]	162 [3] 178 [9] 152 [12]	222 [4] 176 [9] 296 [11]A 276 [12]
G. Runge-Kutta Int.	141074 [2] 228056 [17]	102662 [2] 94960 [3] 176960 [17]	100062 [2] 100042 [4] 117984 [11]A 138414 [17]
H. Linked List Insertion	228 [4] 304 [13] 264 [14]	204 [13] 218 [14] 240 [17]	224 [3] 260 [13] 238 [14]
I. Quicksort	1024 [5] 1008 [6]	14960 [5] 2756 [6]	2968 [5] 1732 [6]
J. ASCII to Float-Pt.	241 [4] 437 [5] 433 [7]	292 [5] 275 [7] 283 [17]	363 [3] 423 [5] 334 [7]
K. Boolean Matrix	832 [3] 909 [6] 896 [8]	582 [4] 776 [6] 932 [8]	384 [6] 566 [8] 640 [17]
L. Virtual Memory Exchange	532 [3] 532 [7] 645 [8]	541 [4] 566 [7] 945 [8]	721 [7] 1058 [8] 780 [17]

INDIVIDUAL S MEASURES

Test Program	Computer Architecture		
	IBM S/370	PDP-11	Interdata 8/32
A. Priority I/O Kernel	216 [3] 286 [12] 742 [14]	48 [4] 32 [12] 32 [14]	26 [12] 28 [14] 26 [17]
B. FIFO I/O Kernel	372 [2] 465 [13] 308 [17]	133 [2] 124 [3] 246 [13]	144 [2] 142 [4] 98 [13]
C. I/O Device Handler	192 [1] 252 [17]	132 [1] 216 [17]	176 [1] 241 [17]
D. Large FFT	454 [11] 454 [9]*	766 [11] 766 [9]*	550 [11] 402 [9] 402 [17]A
E. Character Search	104 [1] 92 [4] 154 [11]	88 [1] 136 [11] 90 [17]	120 [1] 144 [3] 168 [11]
F. Bit Test,Set,Reset	144 [9] 122 [12] 116 [17]	68 [3] 78 [9] 86 [12]	82 [4] 90 [9] 98 [11]A 98 [12]
G. Runge-Kutta Int.	202 [2] 238 [17]	184 [2] 172 [3] 248 [17]	166 [12] 156 [4] 232 [11]A 190 [17]
H. Linked List Insertion	144 [4] 228 [13] 176 [14]	162 [13] 182 [14] 194 [17]	148 [3] 198 [13] 164 [14]
I. Quicksort	340 [6] 407 [5]	940 [6] 1534 [5]	426 [6] 524 [5]
J. ASCII to Float-Pt.	256 [4] 441 [5] 241 [7]	164 [5] 208 [7] 172 [17]	206 [3] 238 [5] 204 [7]
K. Boolean Matrix	224 [3] 267 [6] 284 [8]	174 [4] 232 [6] 284 [8]	156 [17] 130 [6] 180 [8]
L. Virtual Memory Exchange	292 [3] 382 [7] 414 [8]	254 [4] 250 [7] 378 [8]	328 [17] 310 [7] 334 [8]